Modelling programming language semantics using a common semantic model

November 6, 2006

Cong Phuoc Huynh
Supervisors: Mr. Warwick Irwin & Dr. Neville Churcher
Department of Computer Science & Software Engineering
University of Canterbury, Christchurch, New Zealand

Acknowledgments

I would like to thank my supervisors, Mr. Warwick Irwin and Dr. Neville Churcher, for their helpful support and advice to complete this work. I would also like to thank all the fellow Honours students for making this year a memorable experience.

This work is dedicated to my family: my parents and my brother. Without their spiritual and emotional support, this work would not have been possible.

Abstract

Software Engineers are interested in understanding software structure in order to manage its complexity, and to assure and improve its quality. Semantic models can inform tools to communicate this information to humans. The .NET framework provides a common semantic underpinning for a wide range of programming languages in an intermediate language. Nevertheless, there is a semantic gap between the intermediate language and specific programming languages. This research aims to model the mapping between the common semantic concepts and programming language specific concepts. The main benefit of the mapping is to allow us to define metrics, heuristics and visualizations on a set of common semantics and to compare them rigourously across various programming languages.

We propose an architecture for mapping the semantics of multiple programming languages to .NET. In addition, we analyze and design a bidirectional mapping between Java and .NET as part of this architecture. The mapping model is implemented based on existing Java and .NET semantic models, and has been verified on a number of Java and .NET applications.

Contents

1 Introduction

Software is inherently intangible, and usually complex and hard for humans to understand and manage. The process of designing, implementing and maintaining software seeks the optimal compromise between simplicity, functionality, understandability, maintainability, extendability, efficiency and many more attributes. To achieve this, humans need a way to gain insights into the structure of software.

Software engineers usually work with a number of development tools, such as source code editors, which mainly expose the syntax of programming languages, rather than the structure of software. Other tools, such as Unified Modelling Language (UML) diagrammers, are improvements over source code editors, but only provide a limited representation of software structure. The need for gathering information on software structure to inform these tools and communicate them back to the software engineer is an important issue in Software Engineering.

The deep structure of software manifests itself in software semantics, rather than syntax. While semantics represent elements of a program, syntax is the way to express these elements in a particular language. Different syntaxes in different languages can express essentially the same semantics. For example, operations of classes are declared in VB .NET using a different keyword from Java and C#. In object-oriented (OO) languages, semantic entities include elements such as classes, interfaces, fields, methods and other attributes, and the relationships between them such as inheritance, implementation, containment, invocation, and more.

In practice, the developer's mental model of a program is dependent on the languages and tools they use. This is because software semantics are defined differently for each programming language. This makes it hard to compare the semantic features, and to quantify metrics of programs written in different languages.

Recently, the .NET Common Language Runtime (CLR) provides a common semantic underpinning across different languages. In other words, programs written in different languages can interoperate within the same execution environment of .NET. This is achieved by specifying a common Microsoft Intermediate Language (MSIL), which captures the common semantics for a wide range of programming languages, to allow compilation from these languages into MSIL. Language-specific compilers map the syntax and semantics of their language onto these common semantics. Subsequently, the produced MSIL code is able to run on the .NET virtual machine. The compilation process is illustrated in Figure 1.1.

1.1 Research problem

While MSIL captures the semantics at the machine (CLR) level, software engineers work with source code using a specific programming language. Due to the semantic gap between the source language and the common language, the MSIL semantics do not necessarily reflect the exact semantics of a program with respect to a specific programming language. In this research, we introduce a framework which allows a bidirectional mapping between language-specific semantics and the common language

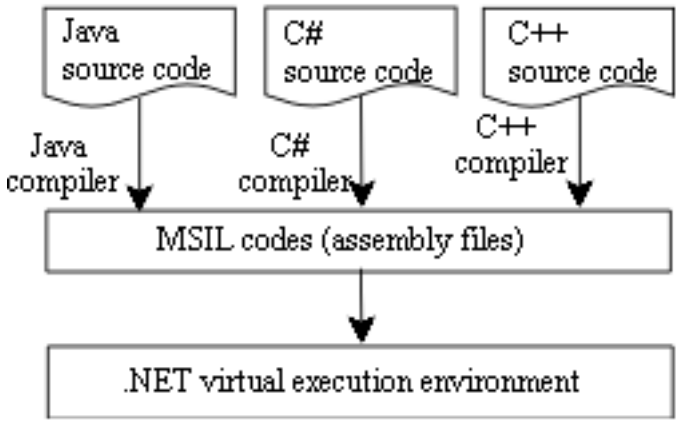

Figure 1.1: Compiling different languages to .NET MSIL.

semantics.

1.2 Motivations

In previous work, a .NET semantic model has been designed to capture the type system defined by the .NET Common Language Infrastructure (CLI) [28]. This language-independent semantic model confers significant advantages. It allows us to define design heuristics, software metrics and visualizations on a set of common semantics and to rigorously compare them across languages.

Software Engineers write programs in specific programming languages. Our semantic mapping approach enables language independent semantics to be managed and integrated with Software Engineering tools for specific programming languages, including Integrated Development Environments, Metric Calculators, Metric Visualizers and other tools.

The heuristics and metrics calculated against an OO language can be translated into another. For example, the Depth of Inheritance Tree (DIT) [4] of a class in J# increases by one when it is compiled into the .NET common language. This is because the top-level superclass `java.lang.Object` in J# is a direct subclass of the .NET library class `System.Object`. The bi-directional mappings between a common model and a language specific model allows the translation of software semantics, heuristics and metrics between different programming languages through the common semantic model, as illustrated by Figure 1.2.

Using a common semantic model takes much less time and effort to map the semantics, heuristics and metrics across different languages than using a direct mapping between any two languages. In fact, with the common semantic model, the expense of mapping language semantics only grows linearly with the number of languages. To date, with an increasing number of languages that can run on .NET, this reduction in time and effort is significant.

1.3 Research Objectives

The following are the specific objectives of this research.

Model the relationship between the common .NET semantics and Java semantics We investigate the feasibility and challenges of mapping the semantics of a specific OO programming language to .NET semantics. Java is chosen as the first language under our study. A prototype mapping model between Java and .NET semantics is developed and verified.

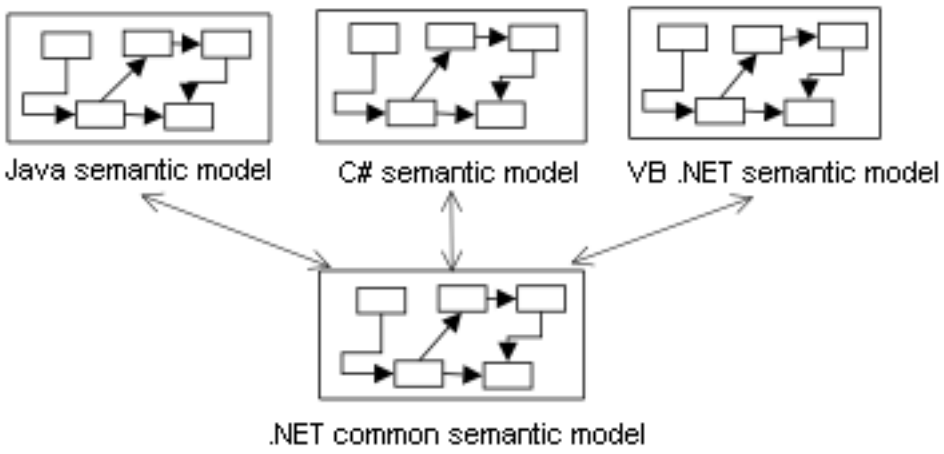

Figure 1.2: Mapping the semantics, heuristics and metrics between multiple languages.

Model and mapping Java generics The latest version of the .NET framework (version 2.0) and Java (1.5) support an important language feature which is generic types. We incorporate this feature into our earlier work on a Java 1.4 semantic model [19] [20] and explore the possibility of mapping them to .NET generics.

Generalize the architecture to accommodate other OO languages We propose an architecture for bidirectional mappings between OO language-specific semantics and .NET common semantics. Although this architecture initially focuses on Java and .NET, it would be designed with possible extension to other languages.

1.4 Report outline

The remaining chapters contain the following contents.

- Chapter 2 provides background information on modelling software structure, the .NET framework and semantic models.
- Chapter 3 introduces our architecture for mapping multiple language specific semantics onto .NET and vice versa, and describes the overview of the mapping model between Java and .NET semantics as part of this architecture.
- Chapter 4 analyzes and designs the mapping model between Java and .NET semantics, including generic type features.
- Chapter 5 presents our implementation approach and the issues related to this approach.
- Chapter 6 presents examples of the bidirectional mapping between Java and .NET to verify the mapping model against our analysis in Chapter 4.
- Chapter 7 discusses the applications and limitations of the mapping model, and mentions future research directions.
- Chapter 8 summarizes the achievements of this research.

2 Background

This chapter presents methods of modelling software structure, the role of the .NET framework in language interoperability and integration and previous work on Object- Oriented semantic modelling.

2.1 Modelling Software Structure

2.1.1 Static Analysis of Software

Static analysis is the act of examining source code of software and possibly other software artifacts, such as UML diagrams, without executing the software. Static analysis aims to obtain information that exposes the static structure of programs before runtime, as opposed to dynamic analysis which focuses more on obtaining the dynamic states of a program at runtime. For example, applying static analysis to a Java program produces a list of its classes, whereas dynamic analysis may result in the approximate number of instances per class.

While static analysis is traditionally used by compilers, it can be used to model software in a representation understandable to humans. By convention, static analysis of source code consists of three phases: lexical analysis, syntactic analysis and semantic analysis. Lexical analysis translates the sequence of characters in a program into a stream of tokens, syntactic analysis develops an abstract representation of the program (also called a parse tree), and semantic analysis analyzes the meaning of the parse tree in terms of the programming language concepts [38]. This research focuses on semantic analysis of software.

2.1.2 Programming language semantics

The term *semantic* in this report indicates the description of the meaning of programming language constructs. Meyer [23] defines it as the runtime effect of the described language construct. Tucker & Noonan 2002 [38] constrains the definition of programming language semantics further, by stating "the semantics of a programming language is a definition of the meaning of any program that behaves identically (with the same input) regardless of the platform on which they run". In brief, the term *semantic* in this report is not to be confused with the semantics of a domain (for example, an account in a banking software).

The semantics of a wide range of object oriented programming languages consist of important concepts such as objects, classes, interfaces, fields, methods and other attributes, and the relationships between them such as inheritance, implementation, containment, method invocation, and more.

Different programming languages may share a number of concepts and relationships. For example, both Java and Eiffel enforce the relationship "every object is a direct instance of only one class". Different syntatic structures used by different languages may express the same semantic concept.

2.1.3 Semantic modelling

Humans understand software by building a *mental model* of programs in terms of their semantic concepts and relationships. The syntactic details are irrelevant to this model. The Unified Modelling Language [30] [11] is a diagramming technique that helps visualizing these concepts and relationships.

Semantic modelling of programming languages focuses on extracting the semantic concepts and their relationships in a program without regard to the syntax used to express them. In other words, semantic modelling is an automatic process carried out by tools to reflect the human mental model of software.

Semantic modelling usually takes inputs from the parse tree representing the syntactic structure of a program. This containment structure obeys the definition of each grammar rule of the language. The semantic concepts are extracted by traversing the parse tree and locating nodes containing semantic elements.

2.1.4 Applications of semantic modelling

Integrated Development Environments

Integrated Development Environments provide a set of tools, including code editors, code browsers, compilers, source code control systems, to develop programs within them. They rely on an underlying semantic model to present the same program in different views and to perform several operations in these views. Every change to the program in a view is passed to a common semantic model, which in turn propagates the changes to all the views simultaneously. In addition, semantic models help IDEs to be intelligent at performing tasks such as checking if a refactoring operation [10] is safe, or highlighting the changes that result in syntactic errors. The Java Development Tool Application Programming Interface (JDT API) [8] is based on a semantic model to provide different software development tools access to the structure of Java programs developed within the Eclipse IDE.

The Collaborative Software Engineering Environment (CAISE) [6] uses an enhanced version of the Java semantic model (JST) [20] to support remote collaboration between software developers. The semantic model is shared in a client/server architecture, and can be modified by several clients simultaneously.

Software Metrics and Visualization

A software metric is a measure of some property of software. Software is usually complex and hard to manage without a quantitative measure. Some early metrics, such as the number of lines of code, or cyclomatic complexity [22], are developed for procedural languages and can be adapted to object-oriented languages. Meanwhile, others are specifically invented for object-oriented software. The metrics suite introduced by Chidamber and Kemerer [4] is a well-known example of object-oriented software metrics.

Software visualization is concerned with visual representation of information about software structure or behavior. The information used for visualization is typically metrics collected from the static or dynamic structure of software, or from software development activities. Visualization is not intended for software quality assurance but can be used to support the understanding and analysis of software systems as well as to manually detect anomalies.

Software metrics and visualization rely on a good source of data to benefit software developers. The software visualization pipeline approach in [5] [20] transforms data in successive stages. The Java [20] and .NET semantic models [28] fit into the semantic modelling phase of this pipeline to

provide rich and comprehensive data on software structure, from which the relevant metrics can be extracted and visualizations can be configured accordingly.

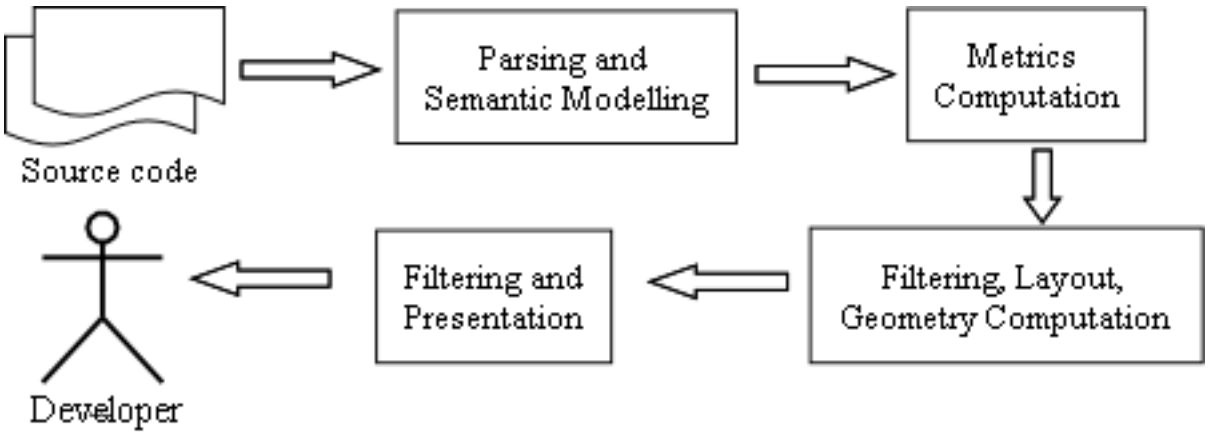

Figure 2.1: The Visualization Pipeline [20].

Although a number of metrics can be computed directly manually from source code, for such as the number of lines of code, or from parse trees such as Weighted Methods per Class [4], the others require a semantic model to obtain an accurate measure. For example, Coupling between object classes (CBO) for a class is the count of the number of other classes which it is coupled [4]. To compute CBO, one needs to know if the methods of a class uses the methods or instance variables of another.

Previous work on a Java and a .NET semantic model has proved to be valuable in providing data for calculating a new metric called CodeRank [29] [28]. CodeRank measures the reusability of software components by ranking each component, such as a class or a method, based on the number of different kinds of links from other components, including inheritance, method invocation, and field reference. Furthermore, the metric can be defined once for the common .NET semantic model and calculated for source code written in any .NET programming language.

Design Heuristics

Design heuristics are guidelines for developers to maintain and improve the quality of their design. Riel [31] provides a list of design heuristics for object-oriented software.

The violations of some heuristics can be detected using a semantic model. For example, the heuristic *limit the depth of inheritance hierarchies to six* is straightforward to quantify for each class in an object-oriented semantic model. Another heuristic “All base classes should be abstract classes.”, although not being quantifiable, can be verified against a program by examining the inheritance relationships.

Heuristics are usually subjectively and vaguely defined. Moreover, they sometimes conflict with each other and there has been a lack of consensus on which heuristics should be adopted. Semantic models allow different interpretations of general heuristics such as “All data should be hidden within its class” by parameterizing and configuring them in terms of the semantic model elements, for example including inherited data items or counting `protected` data as hidden.

2.1.5 Other Approaches to Modelling Software Structure

Semantic modelling is not the only method for modelling software structure. Other methods are Reflection and Abstract Syntax Trees (AST).

Reflection

Reflection is the mechanism to discover contents of types and objects at runtime. The Java standard class library and .NET framework provide reflection APIs to be used by tools for dynamic object creation and method invocation. Apart from these purposes, reflection can be used to extract information about types and their relationships. However, reflection is not designed to be a general static analysis tool to work on a wide range of language features and lacks the following kinds of support.

- Information of private type members is not retrievable by reflection. When modelling the software structure, it is necessary to expose all data, including the private data to software developers.
- It is impossible to retrieve method internals. Although .NET 2.0 reflection provides information on local variables, it does not provide other method internal semantics, including the methods invoked and the fields accessed by a method.

Abstract Syntax Trees

Abstract syntax provides the definition of the essential elements of a grammar, without regard to how they are formed concretely to be syntactically valid [38]. An abstract syntax tree (AST) represents a syntactic component of the source code in each node, possibly with semantic information attached to the node. The links between them represent the semantic relationship between the syntactic components.

Although abstract syntax trees are mainly used by compilers, one can find its application in representing the software structure. For example, programs developed in the Eclipse IDE [8] have their semantic information bound to an AST, so that they can be presented in the user interface. However AST still contains syntactic information, which is not of direct interest to software developers.

2.2 The Microsoft .NET framework

Previous work [28] on semantic modelling has chosen to reflect the semantic concepts in the .NET Common Language Infrastructure (CLI). That choice is motivated by the fact that the .NET CLI provides language-indepedent common semantics. Tools that access data in the common semantic model can be developed once and used for programs written in multiple .NET programming languages. However, there is semantic gap between the common semantics and programming language specific semantics. This research aims to map between the common semantics of .NET and specific programming languages, so that data obtained from the common semantic model can be translated to a particular language for used by software developers.

2.2.1 General .NET architecture

This section gives an overview of the important parts of the .NET Framework relevant to this research. Microsoft has implemented the Common Language Runtime (CLR) based on the CLI Standard [26]. The important components of the CLI are the Common Type System (CTS) and the Common Language Specification (CLS).

The Common Language Runtime (CLR)

The CLR is designed primarily to support a wide variety of programming languages, including object-oriented, procedural and functional languages, to enable language integration. It currently provides strong support for the former two, and minimal support for the latter [17]. A variety of diverse language groups were involved in the design stage of CLR to converge to a design that accommodate to a range of languages. As a result, many compilers currently target the CLR, some of which are produced by Microsoft: C#, J#, managed C++, Visual Basic, and JScript. The remainder, including APL, COBOL, Component Pascal, Eiffel, Haskell# or Mondrian, Mercury, Oberon, Perl, Python, Scheme, and Standard ML, are produced by other companies and organizations [17].

The Common Language Runtime (CLR) provides the underlying infrastructure for the Microsoft .NET Framework. It includes a garbage collector, class loader, metadata engine, and debugging and security services. A main component of CLR is a virtual machine that is able to interpret *managed code* that is compiled from source code in specific languages, into the underlying operating system. Although this virtual machine is thought to be similar to Sun's Java virtual machine and *managed code* is thought to be similar to Java bytecode, they serve the purpose of language independence and cross-language integration while JVM aims at platform independence.

The Common Type System (CTS)

Data types are building block of programs. The simplest types in a programming language are data types, which can be combined to form more complex types. Types describes values and a contract supported by all its values. In some object-oriented languages, two entities belong to the same type if they respond in the same way to the same set of messages. In CTS, types are used to represent entities having same data representation and the same set of behaviours [26].

The Common Type System defines a set of common types and rules for types to be created by compliant languages. CTS-compliant types can be created from the common types by following these sets of rules. These rules enable type compatibility across languages. Examples of the rules in CTS are "all types inherit from the class `System.Object`" and "every reference type can only inherit from a superclass".

The Common Language Specification (CLS)

The Common Language Specification defines a subset of CTS that may be used in code that is intended to be portable between languages. The intention of CLS is a guarantee of language integration while maximizing the number of languages to target the CLR. The set of types in the CLS must be supported by any CLS-compliant consumer or extender. A CLS-compliant consumer is a tool that can use any CLS-compliant type, such as the CLR debugger, while a CLS-compliant extender can use or generate any CLS-compliant artifact, such as the C# compiler [17].

The CLS defines about forty rules to provide a useful subset of the CTS that can readily be implemented in most languages. These rules only apply to "externally visible items", which are type and type members visible or accessible outside their own assembly [26].

Common Intermediate Language and Metadata

Common Intermediate Language As illustrated in Figure 1.1, source code in CLS-compliant programming languages are compiled into a common language, termed Common Intermediate Language

(CIL) by the CLI standard. The Microsoft Intermediate Language (MSIL) is Microsoft's implementation of the CIL. Hereafter, MSIL is used in place of CIL since our discussion specifically involves the Microsoft .NET framework.

MSIL defines a common virtual instruction set that is never actually executed by any hardware. Instead, the CLR defines a stack-based virtual machine to translate these instructions into native code for the underlying processor. Its instruction set closely maps to the abstraction of the CLR's Common Type System. A number of instructions are analogous to operators found in common high-level programming languages.

Metadata CLI-compliant compilers also generate metadata as well as MSIL code and store it with the code in Portable Executable (PE) files. [26] lists the following purposes of metadata:

- A description of the deployment unit, also known as assembly, called the manifest.
- Description of all the types in each module.
- Signatures of methods in each module.
- Enable cross language interoperation.

Metadata stores information about the types defined in the managed code and the external types referenced by the code. According to [26], metadata contains:

- A description of the deployment unit, also known as assembly, called the manifest.
- Description of all the types in each module, including their superclass, superinterface, methods, properties and events.
- Signatures of methods in each module.
- Custom attributes in each module.

Metadata can be read directly from the module that contains it, using a mechanism known as reflection. When a compiler reads a module compiled in a different language, it reads the metadata and performs linking based on the type information it reads.

MSIL and metadata together capture the common semantics defined by the CTS, but not the semantics defined in a particular programming language. The .NET semantic model [28] is populated directly from PE files containing these semantics.

2.2.2 Overview of .NET languages

Among the languages targeting at the CLR [32], Microsoft designs C#, managed C++ (which contains a subset of the type system of C++), J# (with equivalent syntax to Java 1.4) and VB.NET.

C#

C# is a statically typed object-oriented programming language. There is a close relationship betwen C# and MSIL semantics, as commented by Meyer "C# is a human-usable language that directly reflects the .NET object model" [24]. However, C# still has a number of semantics that are not mapped directly to MSIL, such as namespaces, type and type member modifiers.

The design of the C# is influenced by Java language features. However, C# possesses a number of features distinct from Java. For example, type definitions in C# are allowed to be spread over several source files. It also defines the concept of "private interface implementation", and introduces "method hiding", i.e. hiding a method of the superclass with the same signature, to create a specialization point in a method hierarchy [27].

Managed C++

The type system of C++ is tricky to be mapped onto .NET. Therefore, Microsoft only attempts to map a subset of the C++ semantics.

J#

J# is equivalent to Java 1.4 in terms of syntax, but runs on the .NET virtual machine. To support Java in .NET, Microsoft has developed a version of the Java standard class library as a module in the .NET library. To date, this module still lacks important components that the Java library has such as the Swing graphic drawing library. Reflection is also supported in J#, but does not prove to be robust, as discovered when it is used by the J# version of the Java semantic model.

Although the latest version of Java language (Java 1.5) has specified generic type features, Microsoft has not incorporated these features into its current version of J#. This is a motivation for us to model the semantics of Java generics and to map it to .NET generic type semantics.

VB.NET

VB .NET is developed to bring CLR-compatibility to Visual Basic, which is the most popular programming language on the Windows platform. Because it runs on CLR, its language features largely owes to the CLR. In fact, VB.NET is almost the same as C#, except for the syntax [3].

2.3 Semantic models

This section introduces previous work on a Java 1.4 and a .NET semantic model. It includes background information necessary to mapping between Java and .NET semantics.

2.3.1 The Java 1.4 semantic model

Previous work [19] [20] has developed a semantic model (called JST) for the Java 1.4 language. JST identifies the semantic concepts in a Java parse tree and populates a model of these semantics, including packages, classes, fields, methods, constructors, blocks and local variables. By visiting the parse tree, the relationships between these concepts, including inheritance, implementation, containment, method invocation and field reference, are also added to the semantic model. The semantic model is the result of the process shown in Figure 2.2.

A parser generator called Yakyacc is configured with the Java 1.4 language grammar to produce a Java 1.4 compatible parser. It parses Java source code into a parse tree representing the syntactic structure of Java programs. Semantic elements are extracted while visiting the parse tree and are populated in the semantic model. At times, Java source code is not available, for example, some Java libraries are only available in Java bytecode format. In this situation, the semantics are populated using the Java reflection API.

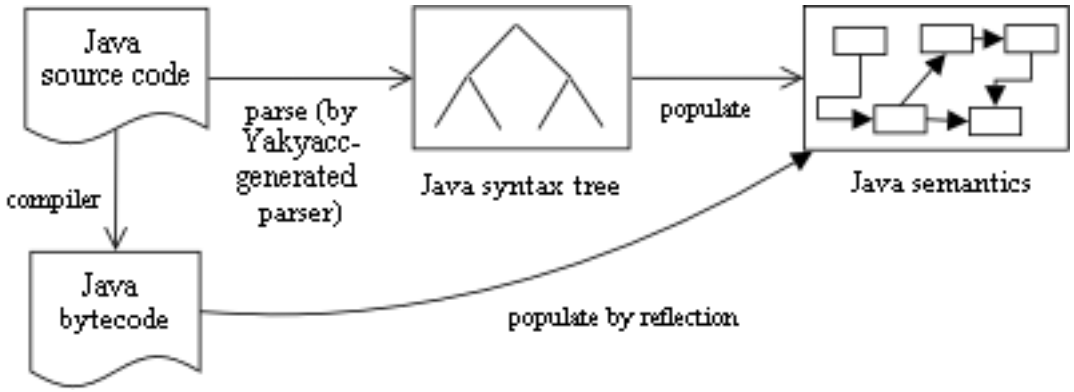

Figure 2.2: Populating JST.

The main limitation of JST is that it only handles applications that are successfully compiled. Compile errors in Java source code result in unresolved links in the semantic model. The other limitation arises in situations when generic type information is not explicitly expressed in Java syntax. For example, if the program has a class `A` associated with a `List` of instances of class `B`, JST cannot identify the type of object contained in the `List` as this is not explicit in the source code.

Since the development of JST, Sun has released a specification of generics in Java 1.5. Currently JST lacks support for modelling Java generics. Modelling Java generics and mapping them to .NET generics are part of this research project.

2.3.2 The .NET semantic model

The .NET semantic model [28] includes the semantics supported by the .NET CLR version 2.0. A number of semantics, including classes, interfaces, arrays, methods, constructors, fields, blocks and statements and modifiers, are also available in Java. This semantic model also includes generics, which is an important language feature of the .NET CLR. The other .NET specific features in this model are assemblies and modules, enumerations, attributes, delegates, events, namespaces, properties and indexers, and a pointer type. In order to capture the common semantics, the semantic model is built from the MSIL and metadata stored in PE files, not from the original source code like JST.

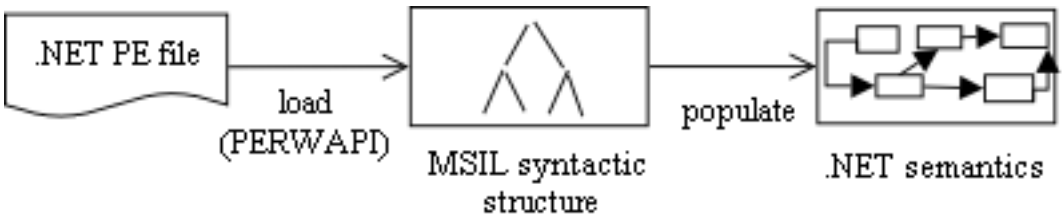

Figure 2.3: Populating the .NET semantic model.

Figure 2.3 shows the process of populating the .NET semantic model from PE files. PE files contain MSIL code that may be compiled from different programming languages in a process shown in Figure 1.1. The Program Executable Reader/Writer Application Programming Interface (PERWAPI) [16] loads the input PE file into a data structure reflecting the metadata and code in it. Subsequently, the .NET semantic model provides a loader to extract relevant data from this data structure into .NET semantics.

The benefit of a common semantic model is that it can capture the core common semantics of any programming language that compiles to the CLR, and is still high-level enough to benefit software developers. In addition, modelling the common semantics allows us to define language-independent heuristics and metrics against the common model, giving us a basis for cross-language comparisons, without loss of rigour.

However, the ability of the common model to represent the semantics of a particular programming language is limited by the semantic gap between that language and the common language (MSIL). Even languages designed for the .NET CLR, including C# and VB .NET, are not exactly mapped to MSIL, as mentioned in 2.2.2. For procedural and functional languages, the gap is even larger.

One of the objectives of this research is to study the semantic mapping between Java and .NET. In this research, their semantic similarities are modelled and implemented as a bidirectional mapping between the Java and .NET semantic models.

3 An architecture for mapping OO languages to .NET

This chapter describes our overall architecture for mapping multiple language semantics to .NET and how the mapping model between Java and .NET semantics is developed as part of this architecture.

3.1 General architecture

Figure 3.1 shows a layered architecture for mapping between language specific semantics and the common .NET semantics. The bottom layer represents the .NET common semantics captured in the .NET semantic model [28]. The middle layer performs mapping between the common semantics and language-specific semantics. This layer is intended to be extensible for different programming languages.

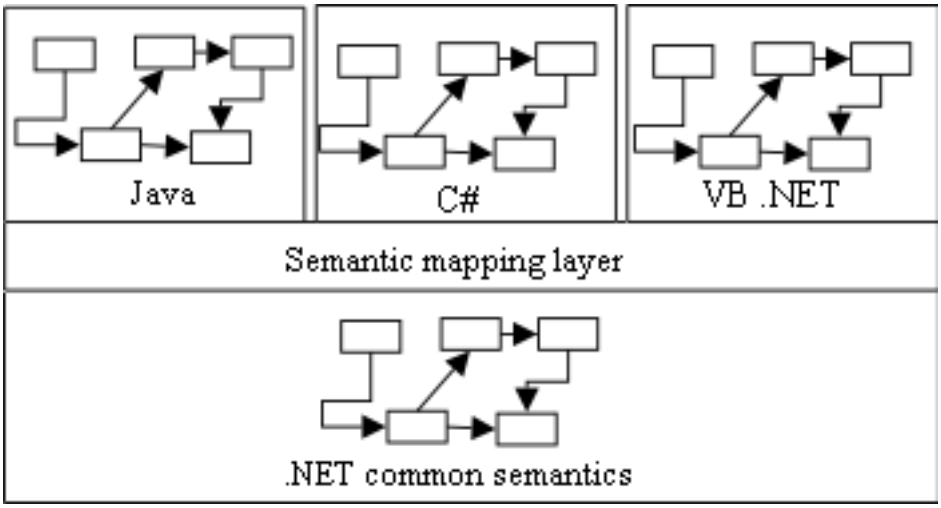

Figure 3.1: An architecture for mapping multiple languages to .NET.

3.2 Modelling the relationship between Java and .NET semantics

3.2.1 Overall design

Figure 3.2 presents a number of equivalent concepts in the Java and .NET semantic models, and the mappings between them. We separate the concerns of mapping these concepts in a mapping model, rather than designing the mappings in the semantic models. This enables the semantic models to be developed and maintained independently of each other.

The mappings are bi-directional, i.e. using the mapping model, one can look up the Java entities equivalent to a given .NET entity and vice versa. In this design, only the mapping model is responsible for responding to these look up queries.

Each mapping between semantic concepts is modelled as a separate entity. The mapping model not only contains one-to-one, but also one-to-many or many-to-many mappings. One-to-many mappings are the case, for examples, when a .NET property is mapped to a group of setter and getter and a number of optional methods in Java.

We decide not to directly model the mapping between relationships such as inheritance and invocation because they can be inferred from the mappings between the entities they relate. For example, the containment relationship between a .NET class and a field implies the containment relationship between the Java equivalents to which they map.

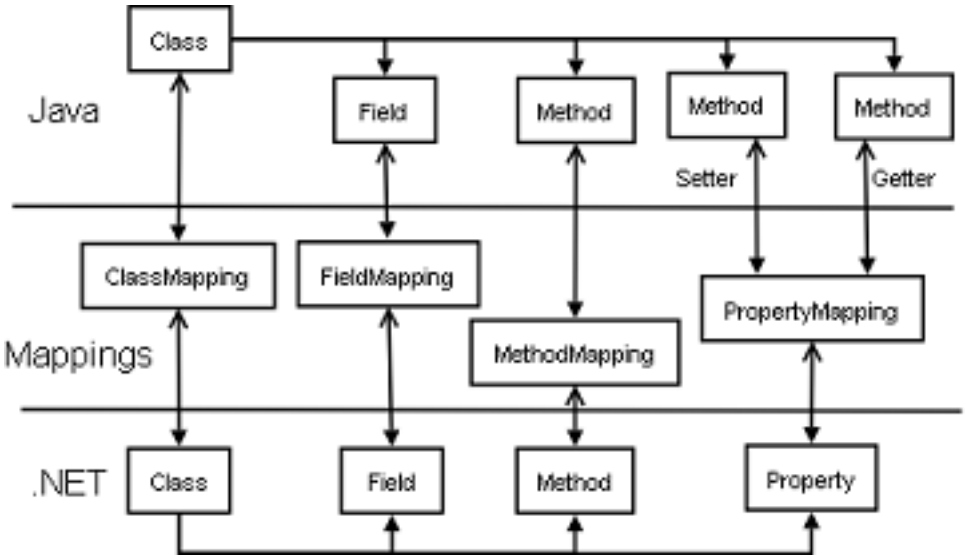

Figure 3.2: A high level design of a mapping between Java and .NET semantics.

3.2.2 Class level design

Figure 3.3 shows the class hierarchy of all the kinds of semantic mappings between .NET and Java. The `Mapping` class defines the common data and behaviours of all kinds of mapping entities. Each `Mapping` is associated with a number of .NET and Java concepts that they relate. Chapter 4 indentifies different kinds of mappings and model them as subclasses of the `Mapping` class.

In addition, a global instance of `MappingCollection` manages all the mappings created in the model at runtime. The operations of adding, removing, and looking up mappings are performed on this singleton `MappingCollection`.

Figure 3.3: The class hierarchy of the mappings between Java and .NET semantics.

4 Analysis and design of the mappings between Java and .NET semantics

This chapter analyzes the similarities and differences between Java 1.5 and .NET semantic concepts. Based on this analysis, the semantic mapping between common features of these languages is designed. A number of semantic concepts can be mapped directly while mapping the others requires more assumptions and complicated manipulation, including introducing semantic elements. Whenever there are different alternatives to mapping a semantic, the simplest one is chosen. On the other hand, the implications of semantic mismatches are described and discussed.

This chapter is organized as follows. Section 4.1 describes the common features of Java 1.4 and .NET. Sections 4.2 and 4.3 respectively describe the language-specific features in Java 1.4 and .NET. Since generics are an important language feature added to the latest specification of the Java language (Java 1.5), the entire section 4.4 is dedicated to them.

4.1 Common semantics between Java 1.4 and .NET

Table 4.1 is a listing of pairs of .NET and Java semantics that are mapped onto each other. In some cases, these concepts do not necessarily have identical meanings in both languages. This section identifies the commonalities as well as subtle differences between these seemingly similar semantics.

4.1.1 .NET assembly vs Java program

In [3], Chappell defines an assembly as an organizational unit of compiled code in .NET. An assembly is logically an indivisible unit containing one or multiple modules. Each module is either a library (.dll) or executable (.exe) file that contains code, or a resource file such as an image. Each assembly has a manifest (metadata) file, which contains information about all of the modules and other files in an assembly.

Assemblies define a boundary for scoping types [26]. A fully-qualified type name consists of the simple type name following the name of the assembly in which the type is defined. Therefore, two different types in two different assemblies are distinct even though they have the same simple name.

Although an assembly is not necessarily executable (it can just be a .dll library), it is regarded as equivalent to a Java program . This equivalence is valid because both concepts are at the top level of the containment hierarchies of both the Java and .NET semantic models. In addition, an assembly can reference other assemblies, in the same way that a Java program references other Java programs. This mapping between .NET assemblies and Java programs is represented as the `Assembly_ProgramMapping` class in Figure 3.3.

Java semantic concept	.NET semantic concept
Java programs	.NET assemblies
Packages	Namespaces
Primitive types	Value types
Classes	Classes
Interfaces	Interfaces
Arrays	Arrays
Instance constructors	Instance constructors
Class initializers	Class constructors
Methods	Methods
Fields	Fields
Setter and getter methods	Properties
Normal code blocks	Normal code blocks
Catch blocks	Catch blocks
Local variables	Local variables
Field references	Field references
Method invocations	Method invocations

Table 4.1: Equivalent semantic concepts in Java and .NET

4.1.2 .NET namespace vs Java package

Namespaces are a way of organizing type names in .NET programming languages such as C# and C++. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system [9]. That means using namespaces helps avoid type name collision between external and internal libraries.

While namespaces are introduced at the source code level in C# and C++ programming languages, they are just a "metadata encoding technique" [26]. At the CLI level, namespaces do not define any scoping semantics. As a result, the concept of namespaces in these languages is lost when source code is compiled into MSIL code. Although not being a MSIL semantic concept, namespaces are included in the .NET semantic model [28] because "they match the mental model of developers".

Namespaces in .NET are essentially similar to Java packages in the way they group types together. However, namespaces are open-ended, i.e. a namespace can be declared in several source files across different modules and assemblies. In contrast, a Java package is only stored in a single directory.

Although not being identical concepts, the strong similarity between these two concepts leads us to relate them via `NS_PackageMapping` class in our design. Based on this mapping, a hierarchy of namespaces is mapped to a hierarchy of Java packages, and .NET types in a namespace are mapped to Java types in the corresponding Java package.

4.1.3 Types

Visibility modifiers

Type visibility specifies the scope in a program where types can be accessed. According to the C# specification [9], "Types declared in compilation units (source files) or namespaces can have `public` or `internal` declared accessibility and default to `internal` declared accessibility". The `public` access modifier means the declared type can be accessed from everywhere, while `internal` restricts

the access to the program declaring the type.

On the other hand, a type member of another type can have any of the five kinds of accessibility: `public`, `protected`, `internal`, `protected internal` and `private`. Section 4.1.4 distinguishes the scopes of these accessibility modifiers.

The rules of type visibility at the Common Language level are slightly different from C#. The CLI specification [26] allows two possible types of visibility for top level types: `public` and `private` (the default). The `private` visibility modifier in CLI restricts type access to the containing assembly, which is different from the `internal` visibility modifier of C#, which restricts type access to the containing program. "Nested types have no visibility, but an accessibility modifier that further refines the set of methods that can reference the type. Because the visibility of a top-level type controls the visibility of all of its members, a nested type cannot be more visible than the type in which it is nested." [26]. The following accessibility attributes are defined for nested types

- `nested assembly` restricts access to the containing assembly.
- `nested famandassem` restricts access to methods of the same type or its subtypes, or types in the same assembly.
- `nested family` restricts access to methods of the same type, or types derived from the contain type.
- `nested famorassem` restricts access to methods of the same type or its subtypes, or types in the same assembly.
- `nested private` (the default) restricts access to methods of the same type only.
- `nested public` no access restriction.

Based on the above descriptions, there is clearly a significant mismatch between the .NET and Java specifications regarding type visibility. The Java language specification [15] defines only four visibility modifiers: `private`, `protected`, `default` and `public`. Java does not define any access scope equivalent to .NET `famandassem`, `famorassem`, and `assembly`. Whenever a .NET access modifier is not equivalent to any Java access modifier, we map it to the Java accessibility modifier with the next wider scope. We observe that the Microsoft J# compiler also widens the accessibility modifiers of types and type members to keep them accessible when they compile to the Common Intermediate Language.

As an example of above approach to mapping modifiers, consider the .NET `assembly` visibility modifier which seems to have an equivalent access scope to the Java `default` modifier. However, they actually define different scopes. Specifically, the `assembly` modifier allows access by classes in a different namespace within the same assembly while the `default` modifier limits access to the same package only. Applying the approach above, we map the .NET `assembly` modifier to the next wider Java visibility modifier, which is `public`.

The mapping between Java and .NET modifiers are summarized in the Tables 4.2 and 4.3.

Inheritance and Implementation relationships

Both Java and .NET mandate that each type, except for `Object`, must inherit from exactly a superclass, and can implement any number of interfaces. This commonality allows straightforward mapping of class hierarchies between these languages. The Java class `java.lang.Object` and .NET class `System.Object` are mapped to each other because they are the top classes of the two hierarchies.

.NET modifier	Java modifier
public	public
private	private
assembly	public
famorassem	public
famandassem	public
family	protected

Table 4.2: Mapping .NET visibility modifiers to Java

Java modifier	.NET modifier
private	private
protected	famorassem
public	public
default	assembly

Table 4.3: Mapping Java visibility modifiers to .NET

The above mapping does not imply an exact mapping between the Depth of Inheritance Tree (DIT) [4] metric across the two languages. Rather, the DIT mapping depends on the implementation of the .NET and Java libraries. The .NET framework implements `java.lang.Object` in J# as a direct subclass of the .NET class `System.Object`. As a result, the DIT of a Java class increases by one when it is ported to J#.

Classes, Interfaces and Structs

.NET classes, interfaces and structs are all represented as the same type of structures in the Common Intermediate Language. However, the .NET [28] and Java semantic models [20] separate these concepts because they differ semantically in the developer's mental model. In both semantic models, they are represented as different subclasses of `UserType`. Therefore, we treat the mapping for classes, interfaces and structs separately.

A class is a semantic unit that incorporates data and behaviours of an object in an OO language. A .NET class may define fields (static or instance), methods (static, instance or virtual), events, properties, and nested types (classes, interfaces, structs and enums) [26]. Meanwhile a Java class can contain fields, constructors, methods, and nested types [15]. Top level classes (classes which are not defined inside another type) in .NET have a one-to-one mapping onto Java . This mapping is represented as the `ClassMapping` class in Figure 4.1.

Interfaces define a contract that other types may implement. ".NET interfaces may have static fields and methods, but they shall not have instance fields or methods. Interfaces may define virtual methods, but only if they are abstract. Interfaces shall not have nested types (interfaces, classes or value types)" [26]. On the other hand, Java interfaces can define inner classes, inner interfaces, constants and abstract methods. Top level Java and .NET interfaces can be mapped one-to-one, and are related by the `InterfaceMapping` class in Figure 4.1.

Like classes, structs may have fields (static or instance), methods (static, instance, or virtual), properties, events and nested types. Although classes and structs are largely similar, there are implicit differences:

- Structs are a kind of value type; they all inherit from the .NET library class `System.ValueType`. When a struct is assigned to another, a new value is created [27]. Unlike classes, structs are not inheritable.
- Structs are stored on the stack instead of the heap. Therefore structs are significantly faster to access than classes [26]. However, structs should only be used to store small data structures since they do not share references to their own data.

Despite these differences, .NET structs are semantically quite similar to Java classes. The mapping between them is represented by the class `Struct_JavaClassMapping` in Figure 4.1.

Our design for mapping classes, interfaces and structs between Java and .NET is shown in 4.1. `ClassMapping`, `InterfaceMapping` and `Struct_JavaClassMapping` are specializations of the more general mapping `UserTypeMapping`, while `UserTypeMapping` provides common mapping operations that are inherited by its subclasses.

This design involves two Gang-Of-Four design patterns [12]: Template Method and Factory Method.

The Template Method Pattern is used to divide the template method `MapType()` in `UserTypeMapping` class into subroutines and to defer their implementations to its subclasses. This method is responsible for creating or looking up a Java or .NET type equivalent to a given one in the other language. An example of a subroutine (also called primitive operation) is `MapJavaToDotNETSuperClass`, which maps the two superclasses of the relevant Java and .NET types. This method is declared abstract in `UserTypeMapping` because only the `ClassMapping`, `InterfaceMapping` and `Struct_JavaClassMapping` knows exactly how to map the superclasses according to the specific types (interfaces, classes or structs) they map.

The Factory Method Pattern is used to declare the contract of abstract methods `MakeNestedJavaType()` , `MakeTopLevelJavaType()`, `MakeNestedDotNETType()` and `MakeTopLevelDotNETType()` in class `UserTypeMapping` for creating a Java or .NET type equivalent to a given one in the other language. These methods are to be implemented by the subclasses of `UserTypeMapping` because only they can know the concrete type of `UserType` to create.

Inner types

The above section assumes that the types being mapped are top-level types. Mapping nested types introduces further complications due to the following mismatches between .NET and Java inner types.

The CLI Standard [26] specifies that:

"Interfaces may be nested inside of classes and value types, but classes and value types shall not be nested inside of interfaces. Nested types are not associated with an instance of the enclosing type. The nested type has its own base type and may be instantiated independent of the enclosing type. However, a nested type (and its members) can access any private (static) members of its enclosing type."

The constraints on .NET inner types conflict strongly with those on nested types in Java. Java allows the following kinds of inner classes: local classes (declared with a name inside a method block), anonymous (declared without a name inside a method block) and member classes. Each of these classes can be declared in a static context (inside static methods and static initializers) or as a static member. "An instance of a non-static inner class is associated with an instance of the immediately enclosing class. The immediately enclosing instance of an object is determined when the object is created." [15]. In constrast, an instance of a static inner class is instantiated independently of the enclosing type and can only access static members of its enclosing type.

Figure 4.1: Mapping types between .NET and Java.

Since .NET nested types are implicitly static, our solution is to map a .NET nested type to a Java static member class. By this approach, the nested type can still access private static members of its enclosing type after being translated into Java. In addition, the Java static inner type will also be instantiated independently of the enclosing type as the original .NET type does.

Java inner classes are mapped to .NET as follows:

- A Java static member class, local class or anonymous inner class declared in a static context is mapped onto a (implicitly static) nested class of .NET.
- Mapping a Java non-static inner type to .NET requires adding the following semantic elements to the .NET semantic model.

  Unlike Java, .NET does not define the relationship between an object created from an inner type and its immediately enclosing instance. We translate this relationship from Java by generating an instance field in the .NET nested type. The field refers to the immediately enclosing instance of the outer class. Access to the members of the enclosing instance is through this field. Because a .NET nested type and its members can access any private members of its enclosing type, changing access modifiers is not required.

  Despite having an instance field to represent the relationship between the instances of the inner and outer types, this method of mapping instances of Java inner types suffers from loss of semantics. Firstly, we cannot translate the constraint in Java that requires each instance of the inner class to be instantiated with an instance of the outer class. Secondly, Java anonymous and local classes would lose the concept of its owner block scope after being converted into .NET. As a result, the .NET inner classes can be used in a wider scope than its Java equivalents.

Array types

An array type in .NET is defined by specifying the element type of the array, the rank (number of dimensions) of the array, and the upper and lower bounds of each dimension of the array [26]. All .NET arrays are derived from the .NET library class `System.Array`.

A language design question for .NET array types is whether arrays should be covariant or invariant. If covariant arrays are supported, an array of a class can substitute for an array of its superclass. Invariant arrays mean that an array of a type is not a subtype of the array of its supertype. CLI supports covariant arrays primarily to allow Java to run on the Virtual Execution System (VES) (partition I, section 8.9.1, page 58–61 [26]).

The other questions are the allowed dimensions for an array, and the allowed array bounds. The CLI supports multi-dimensional arrays, but not zero-dimensional arrays. Each element in a dimension of a multi-dimensional array is an array with the same length. CLI also supports jagged arrays (arrays of arrays). Multi-dimensional arrays are just a special case of jagged arrays. In fact, they are just syntactic sugar in source code. For this reason the .NET semantic model [28] represents all array types as being jagged.

The VES treats single-dimensional, zero-based arrays (also called Vectors) specially. Vectors are distinct from one-dimensional arrays of the same element type with non-zero lower bounds [26]. However, their semantics are just the same as any other kinds of arrays.

The Common Type System (CTS) and Common Language Specifications (CLS) took different appproaches to deciding the lower bounds of array dimensions. CTS allows lower bounds to be specified arbitrarily by the programmer. CLS only support arrays with zero lower bounds for cross-language integration [26].

In brief, the main differences between Java and .NET arrays are:

- Java only specifies arrays with a zero lower bound in each dimension, whereas this is arbitrary in the .NET Common Type System.
- Java only specifies jagged arrays whereas .NET adds the concept of multi-dimensional arrays. C# also defines this concept [27]. Nevertheless, we can always map .NET multi-dimensional arrays onto a Java jagged array of sub-arrays with the same length.
- The inheritance hierarchies between .NET and Java array types are inconsistent. Java arrays implicitly extend `java.lang.Object`, and implement `Clonable` and `Serializable`, and have a `length` field and a `clone()` method. Meanwhile, .NET arrays extend `System.Array` and support cloning, but do not implement `Serializable`.

We map a .NET array to a zero-based Java array and omit the concepts of lower bound information. The remaining information, including the ranks and length of each dimension, are retained because they are useful to software developers. We do not map the supertypes and members of arrays between Java and .NET since their implementations are not consistent across the Java and .NET libraries.

Mapping array types is represented as the `ArrayTypeMapping` class in Figure 4.1.

Built-in types

The .NET CLS defines a set of built-in types which covers all the Java primitive types. The .NET built-in types equivalent to Java are boolean, signed byte, signed short, signed int, signed long, float and double. In the contrary, the unsigned 8, 16, 32 and 64 bit integer types, integer pointer types and `Decimal` in .NET do not have Java equivalents. The mapping between Java and .NET built-in types is represented by `PrimitiveTypeMapping` in Figure 4.1.

The tables 4.4 and 4.5 below summarize the mapping between types from .NET to Java and vice versa. The left column shows the original concept in the source language, while the right column shows the equivalent concept in the target language.

.NET type	Java type
Classes	Classes
Interfaces	Interfaces
Structs	Classes
Nested types	Static member types
Arrays	Arrays
Built-in types	Primitive types (except for signed integers)

Table 4.4: Mapping .NET types to Java

4.1.4 Type members

Accessibility modifiers

The CLI specification [26] defines the following accessibility modifiers for type members.

public Accessible to all referents.

Java type	.NET type
Classes	Classes
Interfaces	Interfaces
Static inner types	Nested (static) types
Non-static inner types	Nested types and enclosing instance references
Arrays	Arrays
Primitive types	Built-in types

Table 4.5: Mapping Java types to .NET

family Accessible to referents in the declaring type and its subtypes.

assembly Accessible to referents in the same assembly.

famandassem Accessible to referents qualified for both family and assembly access.

famorassem Accessible to referents that qualify for either family and assembly access.

private Accessible to referents within the same type.

Amongst these modifiers, `famandassembly` is not translated to any actual modifiers in any .NET languages. In fact, the C# language designers thought `famandassembly` was not important enough to deserve a translation to the source language level.

The mapping of these modifiers to Java has been shown in tables 4.2 and 4.3.

Fields

Java and .NET fields have identical semantics and can be mapped exactly onto each other. Field modifiers other than the accessibility modifiers are `static` and `final`. They are preserved when translating between .NET and Java. Static fields are shared by all instances of the class, while an instance field is only declared for an instance. Final fields are not modifiable after their initialization.

Type operations

.NET specifies three kinds of operations: static constructors, instance constructors and methods. Similarly, Java has three corresponding kinds of operations: class initializers, constructors and methods.

.NET Static Constructors vs Java class initializers A .NET struct or class has an optional static constructor [26]. Static constructors are run exactly once for any given type. It is called before any instances of the class or struct are created or static members are accessed. The constructor is `static`, takes no parameter and returns no value.

Java also defines class initializers for classes, which have all the above properties. Java interfaces do not have class initializers [15]. Although a Java class initializer is composed of several static field initializers, and a `static` block directly inside a class body, it is compiled into a single method in Java bytecode. Similarly, we choose to model a Java class initializer as a single static method in its containing class.

In brief, .NET static constructors are equivalent to Java static initializers.

.NET and Java instance constructors Both .NET Common Language and Java define constructors as operations to create new objects of a class. Therefore they are regarded as equivalent concepts.

.NET and Java instance finalizers Both .NET and Java define a finalizer to be run only once on each object before it is released from memory. This method is called by the garbage collector to close down any resources used by the object [26]. The default finalizer in .NET is implemented in the `System.Object.Finalize()` method, equivalent to the Java finalizer in `java.lang.Object.finalize()` method. Finalizers are considered as a normal methods in our semantic models.

Methods The semantics of .NET and Java methods are essentially the same. The only difference between them can be found in the modifiers of their signatures. The mapping between methods are modelled as the class `MethodMapping` in Figure 4.2.

Method modifiers Section 4.1.4 already describes the mapping between Java and .NET accessiblity modifiers. The Java modifiers `static`, `final` and `abstract` are respectively translated into .NET as `static`, `sealed` and `abstract`.

Below is the .NET-specific modifiers.

virtual To specify that a method of a base type is overridable, it must be marked as `virtual`. Overriding a non-virtual method causes a compile error in C# [27]. Therefore, a non-virtual .NET method is considered `final` in Java.

newslot In addition, if the subclass method definition is marked `newslot` at the MSIL level (or `new` in C#), the subclass method is considered to not inherit from the superclass method, even if the base class method has a matching signature. If the subclass method definition is not marked as `newslot`, it only creates a new virtual method only if there is no `virtual` method of the same name and signature in the superclass. Otherwise, the subclass method overrides the superclass `virtual` method.

Unlike .NET, Java assumes that subclass methods automatically override superclass methods having the same signature and name. The presence of the `newslot` flag in a .NET subclass method definition determines whether it should take a different name from the parent method when it is converted into a Java.

The design of the mapping between Java and .NET operations All kinds of type operations share common data such as modifier, signature and body. Hence, they are represented as subclasses of a common class, `OperationDecl`, in both the .NET and Java semantic models. We model the mapping between operations as a class hierarchy parallel to the one under `OperationDecl` in the semantic models. A design for this mapping is shown in Figure 4.2.

This design also uses the Factory Method Pattern [12] to define the abstract factory method `MakeOperation()` for creating a Java or .NET operation equivalent to a given operation in the other language. The subclasses of `OperationMapping` have the knowledge of the concrete type of Java or .NET `Operation` to create.

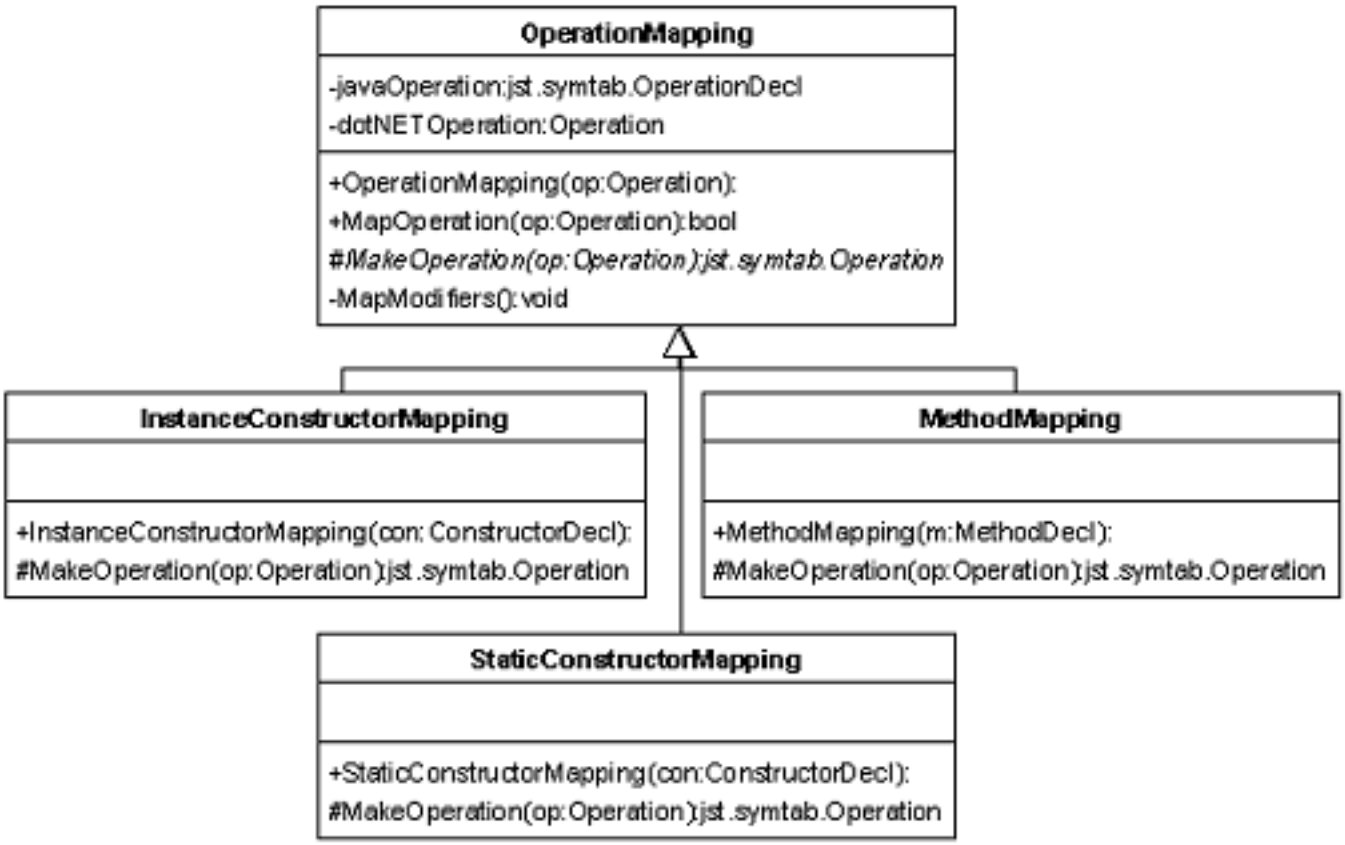

Figure 4.2: Mapping operations between .NET and Java.

.NET properties and indexers

A property defines a named value within a type and the accessing methods for the value. The implementation of the accessors defines how the value is retrieved and stored. By convention, a property defines a getter method and optionally a setter method. Optional methods other than the setter and the getter can be included in the definition of a property.

In C#, a property is syntactically just like a public field. However, unlike a real public field, the containing class of the property controls all the changes made by users through the property [27].

Actually, properties do not add new semantics to a language. The same effect can be achieved by implementing the setter, getter and optional methods that constitute the property. Therefore, a .NET property can be mapped to group of Java methods equivalent to these .NET methods. Conversely, a Java field and its accessor methods can be mapped to .NET in either way: to a property, or a field and a number of methods. Fields and its accessor methods in Java may be related by naming convention, but naming conventions do not necessarily imply the semantic relationship between them. Therefore, we currently map Java fields and methods separately to .NET counterparts, instead of together as a property. The mapping between .NET properties and their Java equivalent concepts is represented in the class `PropertyMapping` in Figure 4.3.

C# defines indexers to index an object with the same syntax for indexing an array. The difference between indexers and properties is only syntax. In fact, an indexer is compiled to a `.property` structure in the metadata, like that of a property. Consider the following C# indexer.

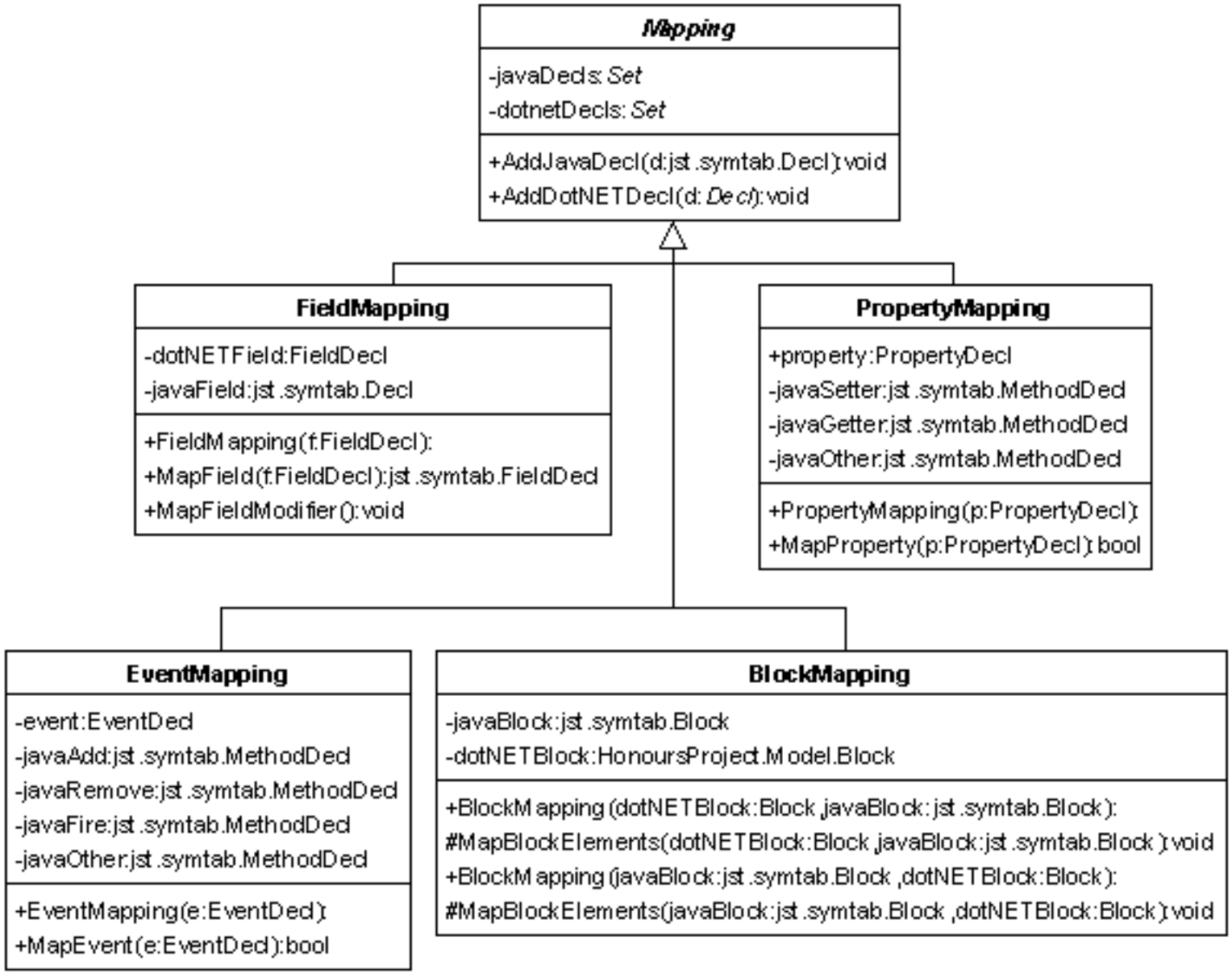

Figure 4.3: Mapping fields, properties, events and blocks.

```
class MyClass {
  string[] myArray = new string[10];
  public string this[int index] {
    get {
      return myArray[index];
    }
    set {
      myArray[index] = value;
    }
  }
}
```

The above indexer is compiled to the following `.property` structure in the CLI metadata.

```
.property string Item
{
  .get instance string MyClass::get_Item(int32)
  .set instance void MyClass::set_Item(int32, string)
}
```

As shown above, indexers are just a special kind of property with an additional parameter in the getter and setter method to declare the index. Indexers are not modelled explicitly in the .NET semantic model [28], and we do not introduce a separate mapping for them.

.NET Events

Events are used to notify class instances of event occurrences. Each event has a delegate type which encapsulates one or more methods with the same return and parameter types. These methods are called event handlers. When the event occurs, all these event handlers are invoked [27].

In C#, events are declared like fields. For example:

```
public delegate void ChangeHandler(object subject, System.EventArgs e);
class Person {
  public event ChangeHandler stateChangeHandler;
}
```

Events are compiled into a MSIL structure which looks similar to that of properties. The structure can consist of methods to add, remove and fire events and other optional methods. The add and remove methods for an event must either be both present or absent [26].

The above event is compiled into the following MSIL code.

```
.event Abduct.ChangeHandler stateChangeHandler
{
  .addon instance void Abduct.Person::add_stateChangeHandler(
                                       Abduct.ChangeHandler)
  .removeon instance void Abduct.Person::remove_stateChangeHandler(
                                       Abduct.ChangeHandler)
}
```

Like properties, an event is mapped to a group of Java methods that comprise it. The design of `EventMapping` is shown in 4.3. Because Java has no definition for events, the reverse mapping is not necessary.

4.1.5 Blocks and intra-block semantics

Scope block and local variables

In Java, a block is a sequence of statements, local class declarations and local variable declarations enclosed in a pair of braces [15]. The scope of a local variable is its immediately enclosing block. Blocks in Java can be nested to form a hierarchy of scopes. Local variables can be referenced only in its immediately enclosing block. Scopes add the semantics about the life time of a local variable. JST [20] models the scope of each local variable as a `Block`.

On the other hand, the concept of a local variable's scope does not exist in MSIL. When a method in a source program (C# or J#) is compiled to MSIL, its internal blocks and statements are flattened into the same block. Hence, after compilation, all the statements and local variables share a single scope.

This compilation process causes a loss of scope semantics at the MSIL level. The .NET semantic model uses the Portable Executable Read Write API (PERWAPI) [16] to extract data from .NET assemblies. Since PERWAPI does not attempt to recover the loss of local variable names and scope information from MSIL, the .NET semantic model does not include them. With the current implementation of JST and the .NET semantic model, the loss of semantics above are unavoidable when Java blocks and local variables are mapped onto .NET.

Exception handling

Both Java and .NET define similar language constructs for handling exceptions, including try blocks, catch blocks and exceptions.

Try and catch blocks in MSIL are determined by the range of instructions they contain. In Java, these blocks are nested in a hierarchy. The .NET semantic model [28] represents try and catch blocks in this way. Nevertheless, it should be changed to model try and catch blocks as ranges of instructions. Mapping of these blocks between .NET and Java will be specified once this changed is made.

Block references

The code in a Java or .NET block can reference fields, create objects and invoke methods. These references are similar semantic relationships between Java and .NET and are mapped by our model.

The mapping between Java and .NET blocks is represented by `BlockMapping` class, as shown in 4.3.

4.2 Java 1.4 specific semantics

The Java 1.4 specific semantics that have not been covered in previous sections are source files and intance initializers.

4.2.1 Source files

Source files are compilation units in Java, i.e. all the types declared in the same file are compiled together. Source files are modelled in JST as the scope of the top level types they contain. Moreover,

they also import types and static members of types from other packages so that these can be referred to by their simple names [15].

The concept of source files only appears in source code; it is lost after the source code is compiled to MSIL.

4.2.2 Instance Initializers

A Java class may have one or several instance initializers outside the scope of any other class members. The instance initializers are declared in a non-static block enclosed in pairs of curly braces directly inside the class body. Instance initializers are executed in the same order as they are declared in the class, just after a constructor of that class calls its parent constructor via super() and before any code in that constructor is executed [15].

This is an example of instance initializers.

```
class X {
  {
    // instance initializer declaration.
  }
}
```

4.3 .NET specific semantics

This section describes the .NET specific semantics that have not been covered earlier.

4.3.1 Modules

A module is a Portable Executable file, either a .dll or .exe files. They consist of one or more classes and interfaces. One or more modules can be deployed as a unit called an assembly [25].

The .NET semantic model [28] models modules as a semantic concept. This decision needs to be reviewed because modules only appear during the build process, they do not define scopes for types, and are not of interest to developers.

4.3.2 Pointer type

.NET Common Type System supports a pointer type to enclose the type systems of programming languages that use pointers, such as C++, whereas Java does not allow this type.

4.3.3 Delegates

A C# delegate serves the same purpose as a function pointer. It is intended to encapsulate one or more methods of the same signature. When a delegate is invoked, the methods it encapsulates are also invoked. Methods can be added and removed from a delegate. In addition, delegates can be composed of other delegates.

In [27] and [26], delegates are regarded as a special type derived from the .NET framework class `System.Delegate`. Each delegate has a number of compiler generated methods to add, remove methods from delegates and to combine and invoke delegates. The .NET semantic model [28] does not model delegates explicitly and treats them as classes.

The Java Language Team [36] does not incorporate delegates into the Java language because they consider Microsoft's "Delegates" harmful. They argue that delegates cause the loss of object-orientation, lack expressiveness, add complexity and are not more convenient than adapter objects.

4.3.4 Enumerations

Although enumerations in Java 1.5 is quite similar to .NET enumerations, our mapping model does not yet include them. Mapping enums is part of our future work mentioned in section 7.3.5.

4.4 Generics

Many modern programming languages support generic types and methods to implement type-safe parametric polymorphism, including polymorphic containers [13] [7].

Generic types and methods are defined with one or more type parameters. They can be instantiated by substituting actual types for their formal type parameters. For example, the Java 1.5 class library provides a `Set` interface with the following definition.

```
public interface Set<E>
```

The generic type above can be instantiated to declare a field as Set of Strings.

```
private Set<String> strings;
```

The scope of a type parameter is the scope of its declaring type or method, including its declaration section. Therefore it can be referred to in the field and method declarations of a generic type.

The advantage of generics is to allow compilers to perform static type checking at compile time. This ensures type safety, for example preventing objects of illegal types from being added to a `Collection`.

4.4.1 Overview of Java generics

Generics have been formalized in Java Specification Request 14 (JSR 14) [2] and incorporated into Java 1.5.

Type erasure

One of the design goals of Java generics is that the bytecode compiled from Java 1.5 source code needs to be backward compatible with previous versions of the Java Virtual Machine. However, the current version of JVM does not support generics. To get around this, the Java compiler uses a technique called type erasure [15]. This technique removes the type parameters of a generic type at compile time and replaces the occurrences of the type parameters with `java.lang.Object`. To preserve the parameterized types of objects, the compiler inserts casting wherever necessary.

The drawback of type erasure is that parametric type information is not available at run time. So one cannot query the parameterized type of an object nor can the JVM check for type violations at runtime [7].

4.4.2 Design of Java generics

This section describes the incorporation of generic types into the Java 1.4 semantic model. Figure 4.4 shows the model of Java generics as a complement to the type system of Java 1.4.

Generic types

A generic type is a type declared with one or more type parameters [15]. Generic types are modelled by adding a field to `UserType` to represent a collection of `TypeParameters` that a generic type has. In our design, non-generic types are a special case of generic type with no type parameters.

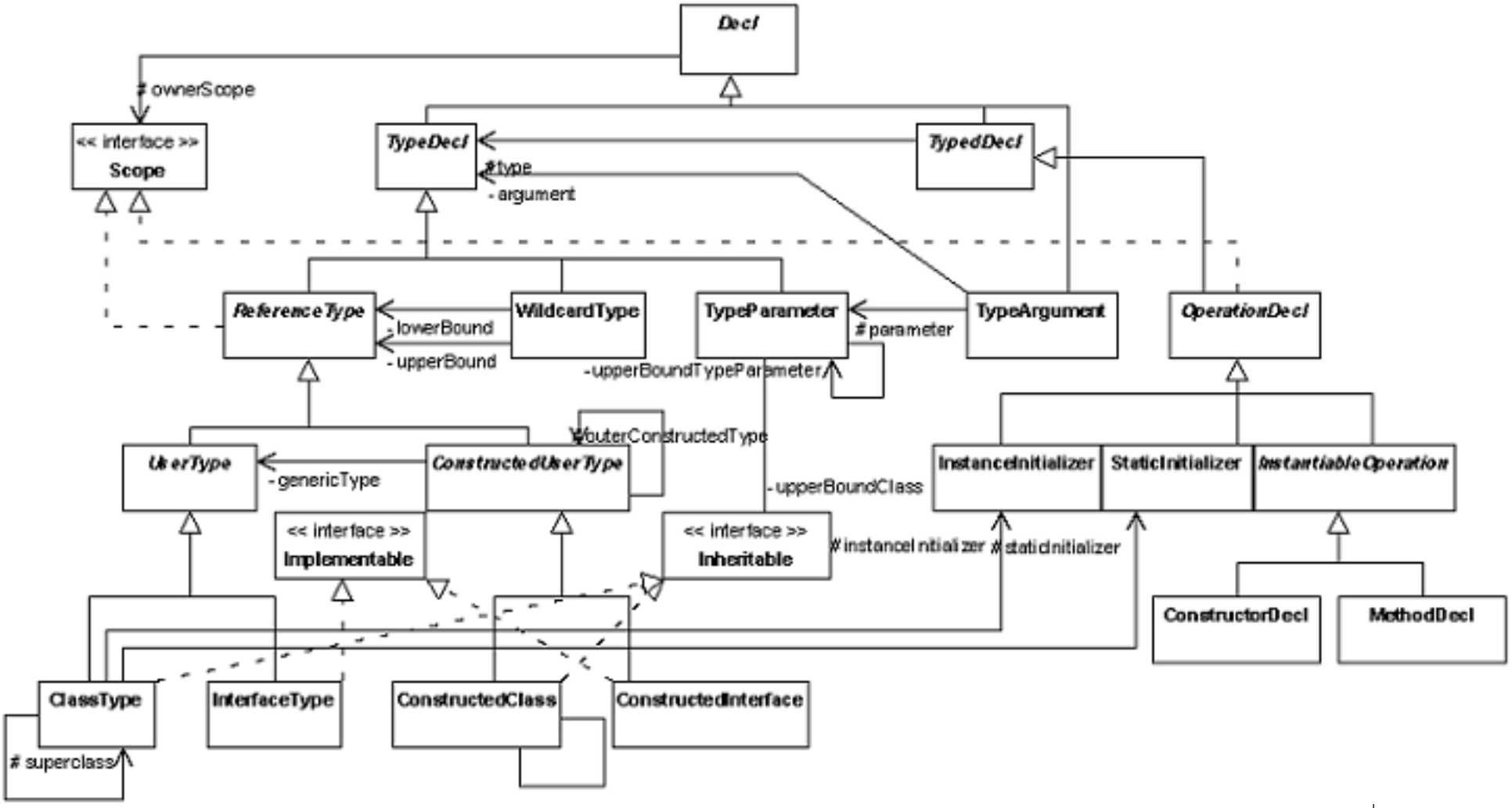

Figure 4.4: Design of Java generics.

Generic constructors

Unlike C# and MSIL, it is possible in Java to declare a constructor as generic, independently of whether its declaring class is itself generic. Similar to generic types, we model generic constructors by adding field representing a collection of `TypeParameters` that a `ConstructorDecl` has.

Generic methods

Similar to generic constructors, generic methods are an enhanced version of non-generic `MethodDecl` with an additional collection of `TypeParameters`.

Type parameters

Type parameters are introduced by generic class declarations, generic interface declarations, generic method declarations, and generic constructor declarations. They define one or more type variables that act as parameters [15]. Type parameters are declared in a section following the type or method name and are separated from the name by angle brackets.

Each type parameter can have optional upperbound types (defined by the `extends` keyword). The first bound in the bound list of a type variable is either a class type or a type parameter. If the first bound is a class type, it can be followed by any number of implemented interfaces. Otherwise no other additional bound is allowed.

The scope of a type parameter is the entire declaration of its declaring type, constructor or method, including the type parameter section itself. Therefore, type parameters can appear as part of their own bounds, or as bounds of other type parameters declared in the same section.

The design of `TypeParameter` is shown in Figure 4.4. The `Scope` of a `TypeParameter` is its declaring type, constructor or method. The bounds of a `TypeParameter` are represented as a `ClassType`, a `TypeParameter`, and a collection of `InterfaceTypes`.

Type arguments

Type arguments define the types used to instantiate type parameters with. In Java, type arguments may be either reference types or wildcards. Unlike Java, C# and MSIL allow type arguments to be value types. Type arguments in JST are modelled similarly to those in .NET. In Figure 4.4, a Java `TypeArgument` references the `TypeParameter` it instantiates and the actual type to instantiate the type parameter with, which can be another `TypeParameter`, a `ReferenceType`, or a `WildcardType`.

Wildcards, denoted by the "?" keyword, are a new construct in the Java 1.5 specification to indicate unspecified type arguments. The main purpose of using wildcards is to "unify the distinct families of classes often introduced by parametric polymorphism" [37].

Unbounded wildcards are useful when the generic class provides functionality independent of the type parameters. However, wildcards should also be able to support polymorphic methods where the method parameter or return types are required to conform to some bound. To satisfy this requirement, wildcards are equipped with bounds to express the range of possible types it covers [37].

Wildcards may be given explicit upper bounds using the syntax `? extends B`, with B as the upper bound, or a lower bound using the syntax `? super B`, with B as the lower bound.

In our design, wildcards are represented as a special type of `TypeDecl` and shown as the class `WildcardType` in Figure 4.4.

Instantiating generic types

Generic types are instantiated by substituting arguments for the type parameters. The arguments may be reference types, type parameters, instantiated types or wildcard type arguments. An instantiated type is model as a `ConstructedUserType` and is created from a `UserType` with a list of `TypeArguments`. The generic `UserType` keeps track of all the types instantiated from it.

`ConstructedUserType` needs to be specialized further into instantiated classes and interfaces, namely `ConstructedClassType` and `ConstructedInterfaceType`. This is because instantiated classes and interfaces have different types of members, for example, instantiated interfaces do not have constructors. The hierarchy of instantiated types is parallel to that of the generic types as shown in Figure 4.4.

The instantiated types and generic types have a number of similar properties that have not yet been captured by this model. Specifically, an instantiated class can be the superclass of another class and an instantiated interface can be implemented by other interfaces. We introduce the common interfaces `Inheritable` and `Implementation` to capture these properties of both instantiated types and generic types. Although we have not explicitly modelled the inheritance and implementation relationships between instantiated types, they can be inferred from their generic types.

Generate instantiated types [14] mentions two ways to generate generic code at run time.

Heterogeneous translation In the heterogeneous translation model, the compiler generates a separate copy of the generic type for each specific instantiation of the type parameters. This is the approach to generate types from C++ templates [35].

Homogeneous translation In the homogeneous translation model, every instantiation of the same generic class shares the same code. Implementation by type erasure in Java [2] implies a homogeneous translation. Type erasure removes parametric type information at run time.

Although the current Java language uses homogeneous translation, the heterogeneous approach is speculated to be more prevalent in the future. A recent research [34] has been successful in retaining parametric type information in Java generic classes and polymorphic methods at runtime to support parametric type-dependent operations. The .NET CLR also has built-in features that support the generation of separate instantiations of generic types without much performance overhead [21]. To support Java generics at the JVM level in the future, we use the heterogeneous translation approach to model them.

The design of instantiated types is shown as the `ConstructedUserType` class in Figure 4.4. The factory method [12] `createConstructedType(List types)` is used to generate a specific kind of `ConsructedUserType` instantiated from a subclass of `UserType`.

Instantiating Generic Constructors and Methods

All constructors, methods and fields of generic types are instantiated at the time the types are created. After instantiation they are the same as members of non generic types. Therefore there is no need to model instantiated operations separately from non-generic operations.

4.4.3 Comparison between Java and .NET generics

The previous section already describes and identifies the Java generics language features that are not present in .NET, including generic constructors and wildcard type arguments. To complete the

comparison between Java and .NET generics, this section lists the .NET specific generic language features.

Java and .NET Generic Interfaces

In Java, it is a compile-time error to refer to a type parameter of an interface anywhere in the declaration of a field or type member of that interface [15]. This is not the case for .NET generic interfaces.

Java type parameter upperbound vs .NET type parameter constraints

.NET defines constraints on type parameters that are not available in Java. For example, it is possible to specify whether a .NET type parameter must have an empty constructor, a reference type or value type.

.NET Generic Structs

.NET structs can be generic. Since .NET structs can be mapped to Java classes, instantiated structs in .NET can be mapped to Java instantiated classes.

Type arguments

.NET allows type arguments to take value types, while Java only allows reference types. On the other hand, .NET does not have any language construct equivalent to Java wildcard type arguments.

Summary

The following table summarizes the similarities and differences between Java and .NET generic features.

Java generics	.NET generics
Generic classes	Generic classes
Generic interfaces	Generic interfaces
_	Generic structs
Generic constructors	_
Generic Methods	Generic Methods
Type Parameters	Type Parameters
Type Arguments	Type Arguments
Wildcard Type Arguments	_

Table 4.6: Compare Java and .NET generic features

4.4.4 Design of a mapping from .NET to Java generics

The mapping from .NET generics to Java is shown in Figure 4.5. The mapping from a .NET generic type as well as its type parameters to Java, is modelled as a concrete instance of `UserTypeMapping`. The mapping of a generic type results in the mapping of its instantiated types.

The mappings between .NET and Java instantiated types are modelled as a subclass of `ConstructerUserTypeMapping`. The factory method `MakeConstructedUserType()` of this class

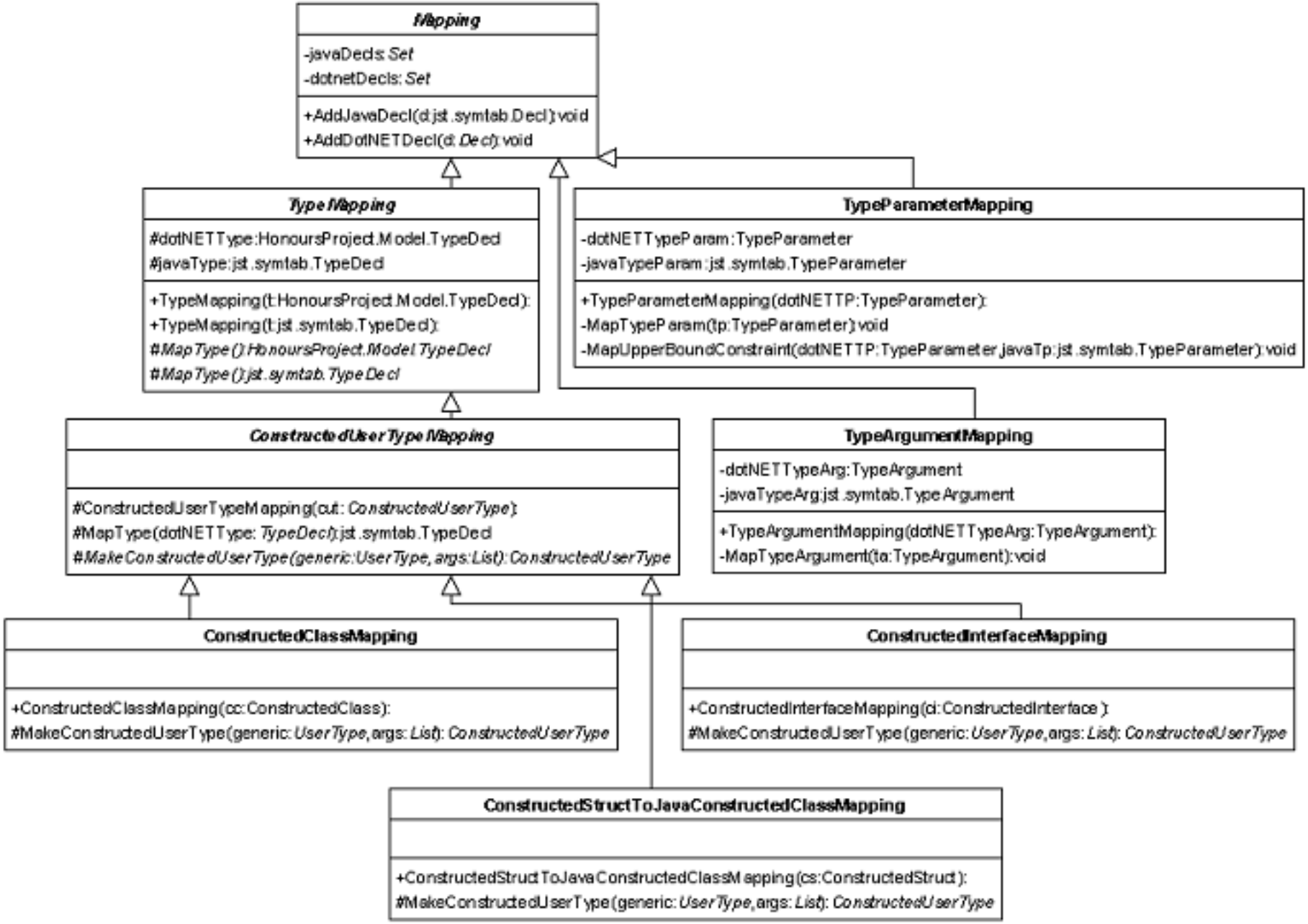

Figure 4.5: Mapping from .NET generics to Java.

is implemented by each of its subclasses to create a concrete Java instantiated type equivalent to a given .NET instantiated type. As discussed before, .NET `ConstructedStructs` are mapped to Java `ConstructedClasses`.

The class `TypeArgumentMapping` is responsible for mapping from .NET to Java type parameters. Amongst different type parameter constraints, we only map the upper bound type constraint of .NET type parameters to Java.

Lastly, the mapping between .NET and Java type arguments is modelled in `TypeArgumentMapping`.

5 Implementation of the semantic mapping

This chapter descibes the implementation of the semantic mapping model discussed in the preceding chapter.

5.1 Updating the .NET semantic model

5.1.1 Compatibility with the Microsoft .NET library

The .NET semantic model was initially developed on the beta release of the Microsoft .NET framework 2.0. A number of changes in the official release of the .NET reflection API has caused several test assemblies fail to load. Corresponding fixes have been added to keep the semantic model funtion in the current version of .NET.

5.1.2 Adding user control

By default, assemblies referenced by an input assembly, other than the .NET framework `mscorlib.dll` assembly, are not loaded into the .NET semantic model. To improve this, user can declare the locations of the referenced assemblies as additional command line arguments when populating the semantic model.

5.1.3 Model improvements

The model of the following features has been revised and modified to conform to the .NET CLI specification.

- A namespace can be shared between different assemblies. Before, the semantic model had a separate namespace for each assembly.
- .NET properties can have more than one optional methods other than the setter and the getter methods. The semantic model used to allow only one optional method for each property.

5.2 Adding Java 1.5 semantics

5.2.1 Supplementary Java 1.4 semantics

The following Java 1.4 semantics have supplemented to the Java 1.4 semantic model.

- Instance initializers
- Class initializers

The parse tree resulting from parsing instance initializer blocks or static initializer blocks in source code contains separate XML elements representing them. Data from all these elements are extracted and combined into a method in the semantic model.

5.2.2 Porting JST to .NET

Before a mapping model between the semantic models is developed, they need to coexist in the same development environment. Originally, the Java semantic model is written in Java, and the .NET semantic is written in Microsoft C# language. We have chosen to port JST to .NET since the .NET CLR allows programs written in different languages to interoperate. Furthermore, Microsoft has designed the J# language, an equivalent of Java 1.4 for the .NET platform. Converting JST from Java to C# takes much less effort than rewriting JST in a different language (such as C#), with only a number of changes to adapt to the J# API.

It is noticeable that the Java version of JST relies heavily of the Java reflection API to look up library classes in Java bytecode. Unfortunately, the J# reflection API is not as robust as its Java counterpart. This limitation can be overcome by replacing reflection with an API to parse Portable Executable (PE) files [16]. The advantage of this API over refection is that it can exploit information of private members and method internals, which cannot be provided by reflection.

5.2.3 Modelling Java generics

The model of Java generics is quite similar to, but not identical to that of the .NET semantic model. The common features of generics between the two languages are the basis to build the model of Java generics. In addition, Java specific features are added to form the the complete model.

Differences in the design of generics between the Java and the .NET semantic models

ConstructedMethod The .NET semantic model includes this class to represent the methods instantiated from a generic method definition. In fact, an instantiated method can be represented as a normal `MethodDecl` with its return and parameter types replaced with the actual type arguments. Hence, the `ConstructedMethod` class becomes redundant. For this reason, the model of Java generics does not distinguish instantiated constructors and methods from normal methods.

Similarly, JST does not model an instantiated constructor separately from a normal constructor.

WildCardTypeArgument A `WildCardTypeArgument` inherits from `TypeDecl` because it represents an unknown type.

5.3 Mapping between .NET and Java 1.4

The mappings between the Java and .NET semantic models are designed in a C# project separate from the J# version of JST and the .NET semantic model. In this design, the mapping model depends on the semantic models. The benefit of this design is to avoid unnecessary coupling between the semantic models, and to allow them to be developed and maintained independently.

5.3.1 Mapping from .NET to Java

Figure 5.1 shows the entire process of mapping a .NET assembly to an equivalent Java semantic model. The Program Executable Reader/Writer Application Programming Interface (PERWAPI) [16]

loads the input PE file is into a data structure reflecting the metadata and code in it. Subsequently, the .NET semantic model provides a loader to extract relevant data from this data structure into .NET semantics. The whole process up to this stage populates the .NET semantic model from the PE file. Lastly, mapping model transforms the .NET semantics into Java semantics.

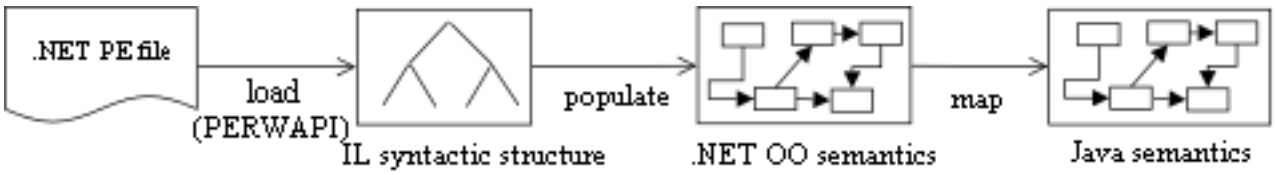

Figure 5.1: The process of mapping from .NET PE files to Java semantics

5.3.2 Mapping from Java 1.4 to .NET

Figure 5.2 shows the process of mapping a Java program to an equivalent .NET semantic model. A Java 1.4 grammar parser, called Yakyacc, is used to parse Java source code into XML files representing the structure of the resulting parse tree [18]. Subsequently, the JST walks this parse tree to extract relevant data for populating Java semantics. The whole process up to this stage produces a semantic model of the Java program. In the last step, the mapping model transforms the Java semantics into .NET semantics.

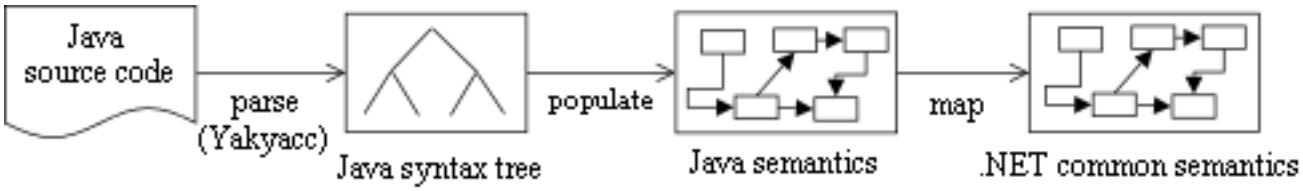

Figure 5.2: The process of mapping from Java source files to .NET semantics

6 Verification

This chapter verifies the conformity of the mapping model to the analysis of the similarities and differences between Java and .NET semantics discussed in chapter 4. The verification process involves populating applications written in either Java and .NET, and mapping the populated semantics to the other language.

We present the results of mapping .NET to Java (excluding generic type features), Java 1.4 to .NET and .NET generics to Java generics in separate tables. Each of these tables contains the mappings of a number semantic elements in sample .NET and Java applications. Each row of the tables is an example for a kind of mapping discussed in chapter 4. The first column is the example index. The second column shows a semantic example in the source language, while the third column one shows the equivalent semantic mapped to a target language.

6.1 Mapping from .NET to Java (excluding generics)

To verify this mapping, the .NET semantic model (which is written in C#) is run on itself to populate its .NET common semantics, and these semantics are mapped to Java. Table 6.1 shows the examples extracted from the result of mapping the non-generic semantics of the .NET semantic model.

These examples show how .NET assemblies, namespaces, classes, interfaces, structs, nested types, fields, instance constructors, methods and properties are mapped to Java. Mapping other language semantic concepts has been tested on other .NET applications.

.NET classes, interfaces and fields in the source assembly are directly mapped into a similar concept in Java, as shown by examples 3, 4, 5, 8. On the other hand, example 2 illustrates semantically similar concepts in .NET and Java with different names, which are Java packages and .NET namespaces. Moreover, example 6 and 7 show that although "struct" is a .NET specific semantic, it is equivalent to a Java class. Example 7 shows that a nested (implicitly static and final) struct in .NET is translated into a static, final inner class in Java. Example 11 and 12 illustrate mappings from a single .NET semantic element (a property) to multiple Java semantics (a getter and a setter methods).

Modifiers are mapped when their declaring types or type members are mapped. In examples 3, 7, 8 and 9, the "public", "protected" and "private" scope of relevant semantic concepts are unchanged after mapping. Meanwhile, the "assembly" modifier in example 10 is widen to "public", which agrees with the approach to mapping modifiers shown in table 4.2.

6.2 Mapping from Java 1.4 to .NET

Table 6.2 shows a number of mappings extracted from the result of parsing and populating the semantics of a Java application (called "Aliens").

While most of the concepts can be mapped directly to .NET, example 7 shows that a Java inner class is translated into a .NET top level class, because .NET does not have the concept of instance inner classes. In addition, the relationship between the original Java inner class

Index	Original .NET semantic	Equivalent Java semantic
1	Assembly HonoursProject	Program HonoursProject
2	Namespace HonoursProject.Model	Package HonoursProject.Model
3	Abstract class public HonoursProject.Model.Decl	Abstract class public HonoursProject.Model.Decl
4	Concrete class public HonoursProject.Model.FieldDecl	Concrete class public HonoursProject.Model.FieldDecl
5	Interface public HonoursProject.Model.Inheritable	Interface public HonoursProject.Model.Inheritable
6	Struct public ClassRankVisitor.ClassRankData	Class public final ClassRankVisitor.ClassRankData
7	Nested struct public MethodsPerClassVisitor.MethodsPerClassData	Inner class public static final MethodsPerClassVisitor.MethodsPerClassData
8	Field private HonoursProject.Model.Decl.simpleName	Field private HonoursProject.Model.Decl.simpleName
9	Instance constructor family ModelVisitor.ModelVisitor()	Instance constructor protected ModelVisitor.ModelVisitor()
10	Method assembly abstract UserType.UpdateConstructedType(...)	Method public abstract UserType.UpdateConstructedType(...)
11	Property public Decl.OwnerDecl	Method public final Decl.get_OwnerDecl()
12	Property public TypeArgument.Parameter	Methods public final TypeArgument.get_Parameter() and TypeArgument.set_Parameter(TypeParameter)

Table 6.1: The result of mapping the .NET semantic model to Java

`java.io.ObjectOutputStream.WrapperObject` with its enclosing class is translated into a field in its .NET equivalent, `java.io.WrapperObject`, to keep a reference to its enclosing instance.

Because "Aliens" does not include array types, Java class initializers and .NET class constructors, this table does not contain examples for mapping these concepts. Mapping them has been verified on other Java applications.

6.3 Mapping from .NET generics to Java

Table 6.3 shows examples extracted from mapping generic semantics of a .NET application to Java. Again, the test application is the .NET semantic model.

These examples include generic types, instantiated types, type parameters and type arguments. The result indicates that although being similar, the model of Java generics and .NET generics are not the same. For example, generic and instantiated structs are .NET language specific features although they can be mapped to Java (in examples 4 and 5).

Remark The results in this chapter show that .NET and Java have a number of identical language features, such as top level classes and interfaces, that can be translated from one language to another and back, without the loss of semantics. Other features, although being similar, translatable between the two languages, but do not preserve their semantics after a round-trip translation to the other language and back to the original language. For example, an original .NET struct is mapped to a Java class and this Java class will become a .NET class in the reverse mapping.

Index	Original Java semantic	Equivalent .NET semantic
1	Program AbductApp	Assembly AbductApp
2	Package Abduct	Namespace Abduct
3	Class Abduct.Person	Class Abduct.Person
4	public abstract class Abduct.Experiment	public abstract class Abduct.Experiment
5	public abstract interface System.Collections.IComparer	public abstract interface System.Collections.IComparer
6	public abstract interface java.util.ListIterator	public abstract interface java.util.ListIterator
7	synchronized nested class java.io.ObjectOutputStream.WrapperObject	class assembly java.io.WrapperObject
8	Field private Abduct.Person.state	Field private Abduct.Person.state
9	Instance constructor public Abduct.Person.Person(String)	Instance constructor public Abduct.Person.Person(String)
10	Method private void Abduct.Person.changeState(int)	Method private void Abduct.Person.changeState(int)

Table 6.2: The result of mapping Java semantics to .NET

Index	Original .NET generic feature	Equivalent Java generic feature
1	Generic class public System.Collections.Generic.List<T>	Generic class public System.Collections.Generic.List<T>
2	Type parameter System.Collections.Generic.List.T	Type parameter System.Collections.Generic.List.T
3	Constructed class List<HonoursProject.Model.TypeDecl>	Constructed class List<HonoursProject.Model.TypeDecl>
4	Generic struct public KeyValuePair<TKey,TValue>	Generic class public KeyValuePair<TKey,TValue>
5	Constructed struct public KeyValuePair<TypeDecl, double>	Constructed class public KeyValuePair<TypeDecl, double>
6	Generic method ClassType.InstantiateType(List<T>)	Generic method ClassType.InstantiateType(List<T>)
7	Constructed method ClassType.InstantiateType(List<HonoursProject.Model.TypeDecl>)	Constructed method ClassType.InstantiateType(List<HonoursProject.Model.TypeDecl>)

Table 6.3: The result of mapping .NET generics to Java

7 Discussion

This chapter discusses the applications and limitations of the semantic mapping model and future work extended from this research.

7.1 An application of mapping .NET to Java semantics

An application of mapping from .NET to Java semantics is to improve the method of populating the semantics of Java bytecode by JST. Figure 7.1 illustrates this process. First, the bytecode is translated into MSIL code, using a compiler such as the IKVM compiler [1].[1] Subsequently, a PE file parser (PERWAPI) [16] is used to extract the syntactic structure of MSIL code from the PE file resulting from the previous step. Then the .NET semantic model populates the common semantics from this syntactic structure. Lastly, the common semantics are translated into the Java semantic model using our mapping model.

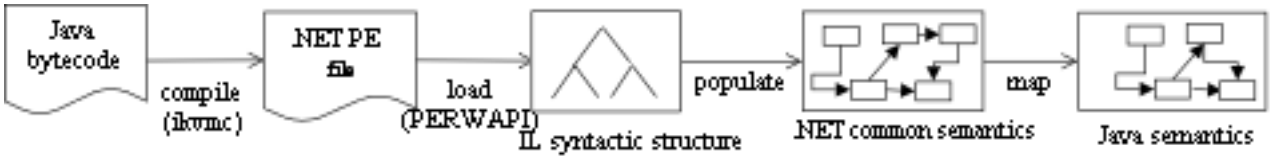

Figure 7.1: The process of populating the semantics of Java bytecode

JST can use the PE file parser (PERWAPI), instead of reflection, to populate the semantics of Java bytecode. PERWAPI has the advantage of being able to provide access to private code and method internals, which reflection cannot.

7.2 Limitations of the semantic mapping process

7.2.1 Reliance on PERWAPI's robustness

The mapping process from .NET to Java, as illustrated in Figures 5.1 and 7.1, is limited by the robustness of each individual component, especially the PERWAPI library. In our experiments on a number of Java libraries containing bytecode, we have encountered run time errors when using PERWAPI to parse MSIL code.

7.2.2 Separation of the semantic concepts of .NET libraries from the original Java bytecode

After Java programs are compiled into .NET, the resulting .NET PE file may contain additional references to .NET libraries. As a results, the referenced libraries are populated as part of the semantic

[1] IKVM compiler is a tool that can compile bytecode contained in Java classes and Java Archive (JAR) files into a .NET assembly.

model of the original Java programs. However, they are not part of the original programs and should be distinguished from the real Java semantics.

7.3 Future work

7.3.1 Reducing the loss of semantics caused by mapping blocks

As discuss in section 4.1.5, our future research should concentrate on reducing the loss of semantics while mapping MSIL instructions to blocks and statements of a particular language. This research direction could start with investigating the approach used by a cross-language code translator that we have discovered, called .NET reflector [33]. It is able to perform code translation from .NET instructions to several programming languages, including C#, Visual Basic, Microsoft Managed C++ and Delphi.

7.3.2 More specific .NET intra-block semantics

Unlike the Java semantic model, the .NET semantic model does not distinguish a field access from a method invocation. In addition, instance creation (using the new() operation), explicit constructor invocations (via a super() call), and local variable accesses are not included in the .NET semantic model. Incorporating these features into the .NET semantic model allows more specific mappings between JST and the .NET semantic model.

7.3.3 Updating Java syntactic parser

Our Java syntactic parser needs to be improved to handle Java 1.5 language syntax, including Java generics. This could be achieved by configuring our parser generator Yakyacc [20] with the additional grammatical rules of Java 1.5 to generate a parser fully capable of parsing Java 1.5 syntax. Generic type features available in Java 1.5 parse trees can be used to verify our design of Java generics.

7.3.4 Mapping Java to .NET generics

At present, we have not analyzed and modelled the mapping from Java generics to .NET. This work can be done after our Java parser has been updated to be compatible with Java 1.5. Using this parser, the semantics of Java generics can be populated from source code and used to verify their mapping to .NET.

7.3.5 Modelling other Java 1.5 semantics

Apart from generics, Java 1.5 also includes enumerations and annotations as new features [15]. Although these features are not as important as Java generics, they should be incorporated into the Java semantic model in the future.

7.3.6 A source code translator

Once the type system of .NET can be mapped to a particular programming language and vice versa, a syntax translator between the two languages can be integrated into our mapping model to perform source code conversion between languages. Although this is not the primary direction of this research, its aim would be practically useful in porting software across languages.

8 Conclusion

In this research, we have demonstrated the need for rigourous semantic models to inform tools of the underlying software structure. The .NET framework provides a common semantics in metadata and MSIL for a wide range of programming languages. By modelling these language-independent semantics, the model can handle programs written in any language that compiles to .NET. However, a common semantic model cannot exactly reflect the semantics of a particular programming language. This research primarily aims to bridge the semantic gap between the common semantics and programming language specific semantics.

We have proposed an architecture for mapping the semantics of multiple programming languages to .NET, and developed a mapping model between .NET and Java semantics as part of this architecture. A number of concepts of Java and .NET are similar and mapping between them is straightforward. Examples of these are top-level and static inner classes and interfaces, fields, methods and constructors. Some .NET specific concepts, such as properties, indexers and events are not available in Java, but can still be mapped to equivalent Java semantics. On the other hand, the largest semantic gap between Java and .NET is found in blocks and statements, Java non static inner types and the .NET pointer type.

Generics are an important language feature that allows parameterization of types, thereby enables type-safe parametric polymorphism. Java 1.5 introduces a number of generic type features similar to .NET generics, including generic types, generic methods, instantiated types and methods, type parameters and type arguments. Apart from these, Java includes features that are not available in .NET generics, including generic constructors and wildcard type arguments. In converse, .NET generics have their own features such as generic structs and type parameter constraints. We have incorporated the model of generics to the Java 1.4 semantic model developed in previous work.

Based on the analysis on the semantic similarities and differences between Java and .NET, our design provides mapping between all the equivalent concepts between them. In addition, the semantic mismatches between these languages may suggest that only a subset of Java language semantics can be included in a .NET version of Java.

We implement our mapping in C#, based on the existing Java (JST) and .NET semantic models developed in previous work. Our design separate the knowledge of mapping from these semantic models so that they are not coupled with each other. For these semantic models and our mapping model coexist in the .NET runtime environment, JST has been ported into J#.

Future research needs to concentrate on reducing the semantic loss resulting from the mapping between Java and .NET semantics, particularly by mapping the semantics of blocks. In addition, the mapping from Java generics to .NET could be modelled and verified once Java generics can be populated directly from source code. This could be done by modifying our Java parser generator to handle Java 1.5 grammar.

Bibliography

[1] IKVM.NET Bytecode Compiler. `http://www.ikvm.net/userguide/ikvmc.html`, n.d.

[2] BRACHA, G., COHEN, N., KEMPER, C., AND MARX, S. JSR 14: Add Generic Types to the Java Programming Language. `http://www.jcp.org/en/jsr/detail?id=014`, April 2001.

[3] CHAPPELL, D. *Understanding .NET: A Tutorial and Analysis*. Addison Wesley, 2002.

[4] CHIDAMBER, S. R., AND KEMERER, C. F. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Engineering 20*, 6 (1994), 476–493.

[5] CHURCHER, N., AND IRWIN, W. Informing the design of pipeline-based software visualisations. In *APVis '05: proceedings of the 2005 Asia-Pacific symposium on Information visualisation* (Darlinghurst, Australia, 2005), Australian Computer Society, Inc., pp. 59–68.

[6] COOK, C., CHURCHER, N., AND IRWIN, W. Towards synchronous collaborative software engineering. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 230–239.

[7] DONOVAN, A., KIEZUN, A., TSCHANTZ, M. S., AND ERNST, M. D. Converting Java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 15–34.

[8] ECLIPSE FOUNDATION, INC. Eclipse Java Development Tools (JDT) Subproject. `http://www.eclipse.org/jdt/`, 2006.

[9] ECMA. *ECMA-334: C# Language Specification*, third ed. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 2005.

[10] FOWLER, M. *Refactoring : improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] FOWLER, M., AND SCOTT, K. *UML Distilled: A Brief Guide to the Standard Object Modelling Language*, 2nd ed. Addison-Wesley, 2000.

[12] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Element of Reusable Object Oriented Software*. Addison Wesley, 1995.

[13] GARCIA, R., JARVI, J., LUMSDAINE, A., SIEK, J. G., AND WILLCOCK, J. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (New York, NY, USA, 2003), ACM Press, pp. 115–134.

[14] GHOSH, D. Generics in Java and C++: a comparative model. *SIGPLAN Not. 39*, 5 (2004), 40–47.

[15] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification*, third edition ed. Addison Wesley, May 2005.

[16] GOUGH, J., AND CORNEY, D. Reading and Writing PE-files with PERWAPI. `http://www.plas.fit.qut.edu.au/perwapi/files/PERWAPI.pdf`, 2005.

[17] HAMILTON, J. Language integration in the common language runtime. *SIGPLAN Not. 38*, 2 (2003), 19–28.

[18] IRWIN, W., AND CHURCHER, N. XML in the visualisation pipeline. In *VIP '01: Proceedings of the Pan-Sydney area workshop on Visual information processing* (Darlinghurst, Australia, Australia, 2001), Australian Computer Society, Inc., pp. 59–67.

[19] IRWIN, W., AND CHURCHER, N. Object oriented metrics: Precision tools and configurable visualisations. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics* (Washington, DC, USA, 2003), IEEE Computer Society, p. 112.

[20] IRWIN, W., COOK, C., AND CHURCHER, N. Parsing and Semantic Modelling for Software Engineering Applications. *ASWEC 00* (2005), 180–189.

[21] KENNEDY, A., AND SYME, D. Design and implementation of generics for the .NET Common language runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM Press, pp. 1–12.

[22] MCCABE, T. J., AND WATSON, A. H. Software Complexity. In *Crosstalk, Journal of Defense Software Engineering* (December 1994), vol. 7, pp. (12):5–9.

[23] MEYER, B. *Eiffel: The Language*. Prentice Hall, 1992.

[24] MEYER, B. The significance of .NET. `http://archive.eiffel.com/doc/manuals/technology/bmarticles/sd/dotnet.html`, 2001.

[25] MICROSOFT CORPORATION. .NET Framework Class Library . `http://msdn2.microsoft.com/en-us/library/ms229335.aspx`, n.d.

[26] MILLER, J., AND RAGSDALE, S. *The Common Language Infrastructure Annotated Standard*. Addison Wesley, 2004.

[27] MOK, H. N. *From Java to C#: A Developers Guide*. Addison Wesley, 2003.

[28] NEATE, B. An Object-Oriented Semantic Model for .NET. Honours report, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, 2005.

[29] NEATE, B., IRWIN, W., AND CHURCHER, N. CodeRank: A New Family of Software Metrics. In *ASWEC2006: Australian Software Engineering Conference, Sydney* (April 2006), J. Han and M. Staples, Eds., IEEE, pp. 369–378.

[30] OBJECT MANAGEMENT GROUP. UML 2.0. `http://www.uml.org/#UML2.0`, 2005.

[31] RIEL, A. J. *Object-Oriented Design Heuristics*. Addison Wesley Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[32] RITCHIE, B. .NET Languages. http://www.dotnetpowered.com/languages.aspx, n.d.

[33] ROEDER, L. Lutz Roeder's Programming .NET. http://www.aisto.com/roeder/dotnet/, n.d.

[34] SASITORN, J., AND CARTWRIGHT, R. Efficient first-class generics on stock Java virtual machines. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (New York, NY, USA, 2006), ACM Press, pp. 1621–1628.

[35] STROUSTRUP, B. *The C++ programming language*, special ed. Addison Wesley, 2000.

[36] THE JAVA LANGUAGE TEAM. About Microsoft's "Delegates". Tech. rep., Sun Microsystems, Inc., 2001. http://java.sun.com/docs/white/delegates.html.

[37] TORGERSEN, M., HANSEN, C. P., ERNST, E., VON DER AHE, P., BRACHA, G., AND GAFTER, N. Adding wildcards to the Java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing* (New York, NY, USA, 2004), ACM Press, pp. 1289–1296.

[38] TUCKER, A., AND NOONAN, R. *Programming Languages: Principles and Paradigms*. Mc Graw Hill, 2002.